UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Politècnica Superior d'Edificació de Barcelona

GRADO DE INGENIERIA EN GEOINFORMACION I GEOMATICA

TRABAJO FINAL DE GRADO

GESTIÓN DE DATOS ESPACIALES PARA LA SIMULACIÓN DE DESPRENDIMIENTOS ROCOSOS

Proyectista: QUIROGA PEREZ, Alejandro

Director/es: NÚÑEZ ANDRÉS, M. Amparo

MATAS CASADÓ, Gerard

Convocatoria: Julio 2019

RESUMEN

El presente trabajo explica los pasos seguidos para la realización de una herramienta en forma de plugin para el software en SIG, QGIS, con la utilización del lenguaje de programación Python, para la automatización de procesos.

En este caso se automatiza la combinación de información vectorial para generar un fichero ráster, que servirá como archivo de entrada para un programa de simulación de desprendimientos rocosos. Más concretamente se leerá información de un modelo ráster,en este caso un MDE (Modelo Digital de Elevaciones), y con las características del mismo se rasterizará y combinarán los ficheros que contiene usos de suelo y situación de barreras protectoras, dando prioridad a estas últimas.

El proyecto es una colaboración con la tesis doctoral 'Caracterización y modelización de desprendimientos rocosos', realizada por el codirector de este TFG, Gerard Matas. En ella se realizan simulaciones de desprendimientos rocosos y las trayectorias que siguen los bloques fragmentados.

En el desarrollo del proyecto se utilizan diferentes tecnologías de la información, así como entornos de desarrollo, destacando la importancia, de que todas las tecnologías utilizadas son de código libre (Open Source).

A lo largo del documento se explicarán los métodos, las herramientas y los datos necesarios para su desarrollo.

ÍNDICE

GLOSARIO

MDT = Modelo Digital de Elevaciones

SHP = ESRI Shapefile, formato de archivo informático propietario de datos espaciales.

SIG = Sistemas de información geográfica

QGIS = Quantum GIS

IDLE = Integrated DeveLopement Environment

TOC = Table of contains “Tabla de contenido”

GDAL = Geospatial Data Abstraction Library

EPSG = European Petroleum Survey Group.

TIFF = Es un format de archive informático para almacenar imágenes de mapa de bits.

TFG = TRABAJO FINAL DE GRADO

LIDAR = Light Detection and Ranging o Laser Imaging Detection and Ranging

1 INTRODUCCIÓN

En el mundo del SIG (Sistemas de Información Geográfica) ha dado un gran avance referente a las tecnologías, debido a la gran cantidad de datos geográficos existentes. Hoy en día cualquier persona del mundo utiliza datos geográficos, para buscar un destino de viaje, un restaurante cerca de su hogar, la ruta más óptima para desplazarse, etc.

De tal forma se requería la necesidad de un visualizador para comprender dichos datos, 'más vale una imagen que mil palabras'. A partir de esta necesidad se han creado un sinfín de herramientas entorno a dicha información. Como pueden ser, bases de datos donde almacenar y administrar la información, visualizadores donde representarla, incluso herramientas donde poder gestionar y analizarla.

La tecnología de código abierto (Open Source), ha sido un gran impulsor de este tipo de tecnologías, donde un mayor número de personas han podido acceder, experimentar y trabajar, con lo cual se ha creado una comunidad estable alrededor de los Sistemas de Información Geográfica.

Debido a la gran cantidad de información que se genera, surgió la necesidad de crear programas que automaticen procesos, con el objetivo de sustituir rutinas manuales, acelerando el tiempo de ejecución de las tareas y eliminando los posibles errores humanos. Por tal motivo surge la realización de este proyecto.

El proyecto es parte de una colaboración con la tesis doctoral "Caracterización y modelización de desprendimientos rocosos" desarrollada por uno de los directores de este TFG, Gerard Matas, la cual estudia las trayectorias que describen los bloques rocosos que se desprenden incluyendo la fragmentación.

Para llevar a cabo dichas simulaciones es necesario disponer de unos datos de partida de los cuales se encuentran MDT y tipo de suelo por los que se desplaza junto con las barreras que puede encontrar.

Estos procesos de simulación requieren la realización de múltiples ensayos con diferentes datos de partida respecto a tipo de terreno y diferente disposición de obstáculos, por lo que es conveniente crear una herramienta que agilice la creación de esta información inicial.

El proyecto que se desarrolla se encargará de crear un plugin para QGIS, mediante la escritura de código en el lenguaje de programación Python, que permita, a partir de unos datos de entrada, que se componen de datos vectoriales y datos ráster, la creación de un fichero con información combinada entre los datos, cuya finalidad será utilizar los datos resultantes en las simulaciones.

Para hacer más cómoda dicha tarea dentro del plugin, el usuario únicamente deberá entrar los datos, y el código detrás de este, ejecute los procesos necesarios para generar un archivo de salida, que sea entendible y legible para su análisis.

1.1 objetivos

El objetivo principal que se quiere conseguir en este proyecto es la automatización de procesos mediante código, para facilitar y agilizar los múltiples casos en los que se pueden dar un desprendimiento rocoso.

Específicamente la creación del modelo ráster de usos del suelo válido para la simulación, resultado de la combinación de varios datos en los que se encuentran datos vectoriales (SHP) y datos ráster.

Para que el proceso sea transparente al usuario se creará un Plugin, para a QGIS, cuya obtención de datos, sean válidos como archivos de entrada para la simulación en el software RockGIS, desarrollado en la tesis anteriormente citada.

Como objetivos secundarios aprenderemos un lenguaje de programación, que sea capaz de interactuar con un Sistema de información Geográfica, conocer diferentes entornos de trabajo y facilitar al usuario el manejo de las tecnologías.

En una pequeña parte también aprenderemos como se desarrolla y se gestiona un proyecto de tal envergadura a nivel nacional, como puede ser RockModels.

2 Antecedentes

Un desprendimiento rocoso es un fenómeno de inestabilidad gravitacional que se produce cuando una masa rocosa se separa de una vertiente y se propaga mediante una caída libre y/o rebotando y/o rodando. Durante la propagación puede haber fragmentación. Este fenómeno puede suceder de manera aislada o de manera conjunta. Pese a su reducido volumen en comparación con otros movimientos de tierra, los desprendimientos pueden ser más destructivos, debido a sus altas velocidades, y consecuentemente a las altas energías que pueden adquirir durante su propagación.

Debido al riesgo que supone este tipo de eventos geológicos se han planteado desde el departamento de ingeniería civil y ambiental de la UPC varios proyectos de investigación que ayuden a conocerlos más profundamente. El primero de ellos fue RockRisk (2014-2016) antecesor del actual proyecto RockModels (2017-2019) en el cual se enmarca este TFG. En la siguiente sección se explicará con más detalle este proyecto.

2.1 Rockmodels

RockModels es un proyecto de investigación financiado por Ministerio de Economía, Industria y Competitividad del Gobierno de España, cofinanciado por la Agencia Estatal de Investigación (AEI) y el Fondo Europeo de Desarrollo Regional (FEDER, 2017-2019) referencia BIA2016-75668-P, el proyecto propone abordar los siguientes objetivos: [8]

- Identificación explicita de volúmenes de roca inestable y evaluación de la estabilidad.
- Puesta a punto y validación del modelo de fragmentación de desprendimientos rocosos.
- Análisis de la propagación y puesta a punto del modelo RockGIS para desprendimiento rocosas, incluyendo la fragmentación.

RockModels engloba diferentes herramientas para predecir y prevenir los desprendimientos rocosos, y se desarrollarán diferentes algoritmos y técnicas para el seguimiento de la caída de bloques en ensayos controlados.

Debido a la falta de datos de campo detallados de sucesos de desprendimientos, la única manera para poder hacer una estimación correcta de las trayectorias de los bloques, es mediante modelos de propagación de bloques. Por este motivo se ha desarrollado la herramienta RockGIS, que se encarga de definir las posibles trayectorias de los desprendimientos, en estas, la información referente a la velocidad del bloque, alturas de rebote, así como la distribución espacial, es básica para el correcto diseño y verificación de las medidas de protección.

2.2 RockGis

RockGis es una herramienta de simulación tridimensional de desprendimientos rocosos en un entorno GIS. Este programa incluye un módulo de fragmentación que permite considerar este fenómeno en las simulaciones para mejorar los análisis de peligrosidad necesarios para la cuantificación del riesgo por desprendimientos rocosos. [1]

La aportación del proyecto, objeto de esta memoria, se centra en esta herramienta. Como se ha mencionado anteriormente, el objetivo es crear un plugin que proporciona el terreno con las características necesarias para desarrollar las simulaciones.

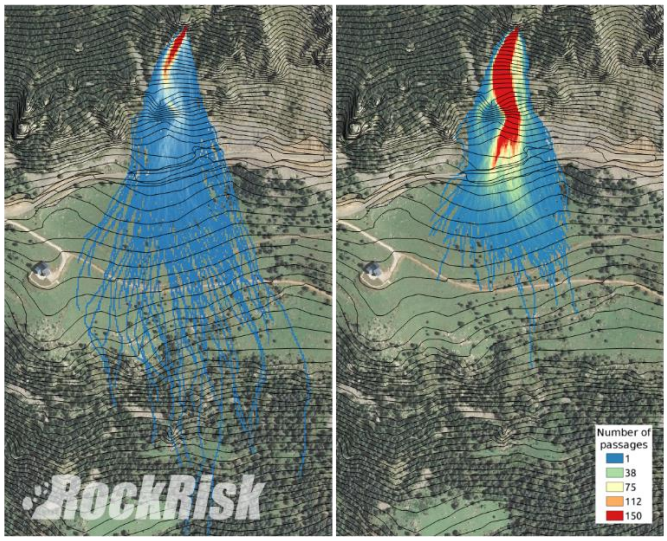

***Figura 1**. Simulación de desprendimientos rocosos [1]*

3 ENTORNOS DE TRABAJO

Existen diferentes y variadas definiciones de Sistemas De Información Geográfica (SIG). Una de ellas nos dice que es una integración organizada de hardware, software y datos geográficos, con el fin de almacenar, manipular, analizar y desplegar la información geográfica referenciada.[2] Según el Centro Nacional de Información Geográfica y Análisis (NCGIA, por sus siglas en inglés):

'Un SIG es un sistema de hardware, software y procedimientos elaborados para facilitar la obtención, gestión, manipulación, análisis, modelado, representación y salida de datos espacialmente referenciados, para resolver problemas complejos de planificación y gestión.'

En nuestro caso nos centraremos en estas últimas funciones "*resolver problemas complejos para* planificación y gestión". RockGIS tienen este objetivo, analizar cuál será la trayectoria de los bloques que caen desde una pared rocosa. Para ello se deben llevar a cabo una serie de pruebas cuya automatización facilitaría y optimizaría enormemente el trabajo.

Se ha de tener en cuenta el terreno por donde se desplazará el bloque, tanto la orografía como el uso del suelo, no es lo mismo moverse por cultivos que por un prado, y los obstáculos que encontrará en su recorrido. Por ello será necesario crear una nueva capa geográfica que nos aporte todos estos datos.

Todo el trabajo de creación de esta nueva información de forma automática, se establece en el entorno de SIG, por ello es necesario introducirnos en el uso de datos y programas con los cuales interactuamos a lo largo del proceso.

3.1 QGIS

Como mencionamos en el apartado anterior, dentro del conjunto de software de un SIG existen diferentes herramientas con las cuales podemos manejar los datos geográficos, herramientas como pueden ser ArcGIS, gvSIG y QGIS, siendo este último utilizado para el desarrollo del proyecto.

QGIS (Quantum GIS), es un SIG de código libre desarrollado en C++, usando la biblioteca QT para su Interfaz Gráfica de usuario, una de sus grandes ventajas, es que trabaja en cualquiera de los sistemas operativos como GNU/Linux, MAC OSX, Windows y Android (en fase experimental). [3]

Permite manejar formatos ráster y vectoriales a través de las bibliotecas GDAL y OGR, así como bases de datos, algunas de sus características son:

- Soporte para la extensión espacial de PostgresSQL, PostGIS.
- Manejo de archivos vectoriales Shapefile, DXF, DWG, etc.
- Soporte para tipos de archivos raster (GeoTIFF, TIFF, JPG, PGN, etc.)
- Soporte para la conexión de diferentes servicios remotos como WebMapService, WebTileService, WebFeatureService, WebCoverageService, etc.

Una de las razones por la que se ha escogido QGIS para la realización del proyecto, entre otras, por ser un software de código libre (Open Source), puede ser modificado libremente de manera que se pueda realizar diferentes y más especializadas funciones. El usuario puede adicionar muchas funcionalidades escribiendo sus propios complementos, estos pueden ser escritos en C++ o Python.

La primera versión de QGIS fue lanzada en Julio de 2002, de la mano de Gary Sherman en su búsqueda de un visor SIG para Linux, a lo largo del tiempo ha ido evolucionando y mejorando, hasta convertirse en un proyecto de incubadora de la Open Source Geospatial Foundation (OSGeo) en 2004, impulsado por su comunidad, hasta la fecha de febrero de 2018 que se lanzó la nueva versión de QGIS 3x, basado en Qt5, PyQt5 y Python3, versión utilizada actualmente en el proyecto.

3.2 Python

Python fue creado a finales de los ochenta por Guido van Rossum, en el Centro para las Matemáticas y la Informática, en los Países Bajos. Como curiosidad el nombre del lenguaje proviene de la afición de su creador por el grupo humorista Monty Python. [4]

Python es un lenguaje de scripting y orientado a objetos de creciente expansión y popularidad derivada por su simplicidad, versatilidad y rapidez de desarrollo. Python ofrece muchas ventajas, es fácil de aprender, es estable, multiplataforma, escalable, y lo más importante es de código abierto (Open Source), con lo cual tiene una gran comunidad de usuarios de manera que ofrece una gran cantidad de información documentada y ejemplos disponibles en la red.

Como se menciona anteriormente, el usuario puede crear complementos y funcionalidades, a través de Python, creando scripts. El acceso a QGIS desde Python es posible gracias a la utilización de bindings.

Un binding es un módulo Python, escrito en lenguaje C++, que permite realizar llamadas a funciones en C++ desde un script hecho en Python. Se puede ver como un "puente" desde C++ a Python.

Por lo tanto, PyQGIS es el conjunto de bindings que el proyecto QGIS aporta para que el desarrollador acceda a las librerías de QGIS (C++) a través de Python.

Utilizando PyQGIS, podemos acceder a diversas bibliotecas de QGIS como pueden ser:

- Core: Contiene las funcionalidades GIS.
- Gui: Contiene controles para la interfaz de usuario.
- Qt: Contiene funcionalidades para su interfaz gráfica.
- MapComposer: Contiene funcionalidades de salidas gráficas.

3.3 Qt Designer

Qt Designer es la herramienta para diseñar y crear interfaces gráficas de usuario (GUI), a partir de los componentes de Qt. Puede componer y personalizar sus ventanas o cuadros de diálogo. [5]

Qt es un marco de desarrollo que provee clases para dotar a determinadas aplicaciones de interfaz gráfica, sus clases están escritas en C++.

PyQt5 son los bindings para la biblioteca gráfica Qt5, que permiten hacer uso de las clases y objetos de interfaz de usuario del proyecto Qt5 desde Python.

Los formularios y Widgets (pequeña aplicación o programa) se integra a la perfección con el código programado, utilizando el mecanismo de ranuras y señales de Qt.

3.4 Herramientas de Desarrollo

Para el manejo de grandes cantidades de líneas de código, es necesario una IDLE, para mantener el código ordenado y claro. Una IDLE es un entorno grafico de desarrollo de software para programadores y desarrolladores.

Es la manera que tenemos de comunicarnos con el programa de manera extendida, añadiendo más posibilidades de las que nos ofrece el mismo.

Durante el proceso se han utilizado varias IDLE, las más importantes son la IDLE de Python que incorpora QGIS, y Sublime Text.

La IDLE de Python, es más funcional ya que inmediatamente podemos ejecutar el código dentro del mismo y visualizar los resultados, comprender los errores y realizar modificaciones sin cambiar de aplicación.

La siguiente IDLE es más confortable a la hora de trabajar, ya que su estructura y la diferenciación de líneas de código es clara, utilizando colores según las características que tiene la sintaxis.

4 DATOS DE ENTRADA

Para llevar a cabo el proceso es necesario unos datos espaciales, estos presentan dos tipos de propiedades, las geométricas y las descriptivas, siendo estas las que proporcionan su utilidad.

Propiedades Geométricas: todos los datos espaciales están vinculados con un lugar, lo que se conoce como la 'georreferenciación'. Esta vinculación se realiza mediante coordenadas que definen la localización de puntos, líneas o áreas.

Propiedades Descriptivas: es la característica que representan (usos del suelo de un territorio, tipo de carretera, etc.), estas características pueden ser muy diversas, desde valores numéricos hasta documentos gráficos. [6]

Las fuentes donde se extraen los datos son muy diversas como pueden ser planos existentes en formato papel, información estadística, como reutilizar la información generada por un proyecto de ámbito territorial (carreteras, canales, etc.). Por otro lado, cuando se requieren análisis territoriales más específicas es necesario generar los datos, ya que en muchos casos no se encuentra disponible. Hoy en día existen muchas técnicas con las cuales poder generar información geográfica de calidad.

Para llevar a cabo las simulaciones de trayectorias de bloques fragmentados en RockGIS es necesario disponer de unos datos espaciales de partida entre los que se encuentran su tipología y distribución, en este caso, MDT por el que se desplaza, tipo de suelo por los que se desplazan los bloques y barreras que impedirían su avance. (**Figura 2**)

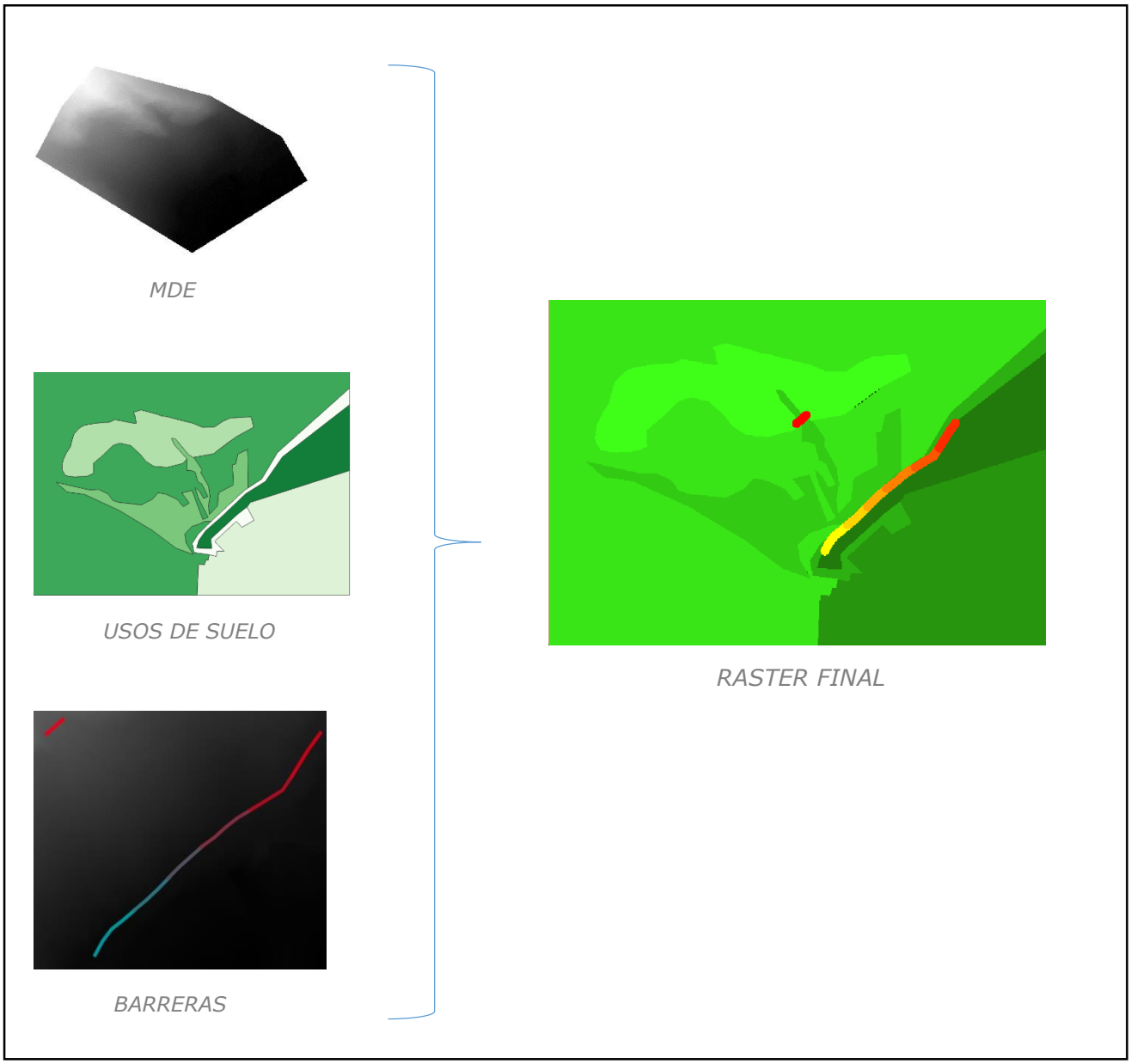

***Figura 2**. Esquema de datos de entrada utilizados*

Los datos de partida necesarios, suelen encontrarse en formato: ráster, y vectorial. Pasaremos a describir a continuación cada uno de ellos.

4.1 Datos vectoriales

El proyecto contiene información vectorial sobre:

- Usos del suelo (**Figura 3**). En este caso la geometría que definen los diferentes usos son polígonos que diferencian cada zona. En el ejemplo que se va a desarrollar, los usos del suelo que encontramos son:
    - Carretera
    - Edificios
    - Roca
    - Tartera
    - Vegetación Densa
    - Zona Urbana

  Hay que diferenciar los tipos de suelos por los que se desplaza. Ya que no tiene el mismo comportamiento el deslizamiento teniendo en cuenta tipo de suelo.

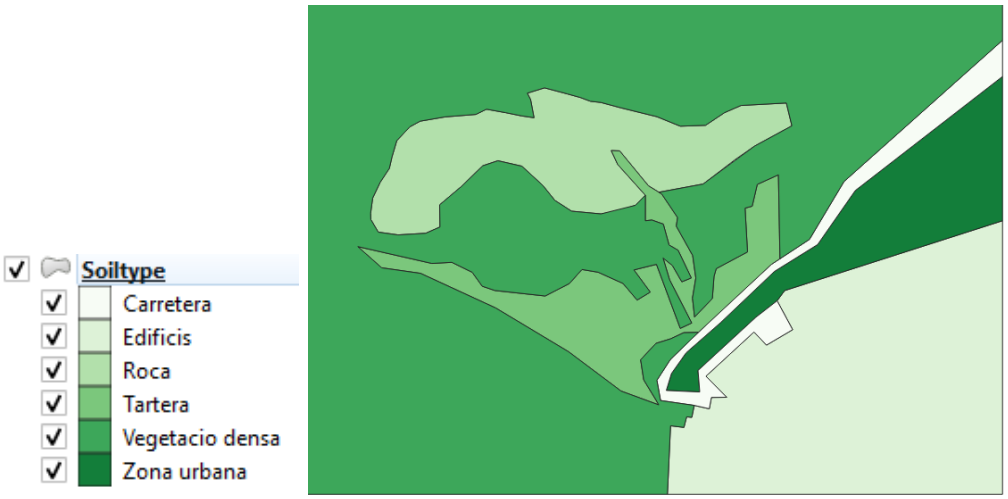

***Figura 3**. Información poligonal correspondiente al uso de suelos, a la derecha se muestra la leyenda, donde cada tipo de uso se representa con un color.*

- La ubicación de barreras protectoras (**Figura 4**) en formato lineal, estas sirven para minimizar los daños causados por los desprendimientos.

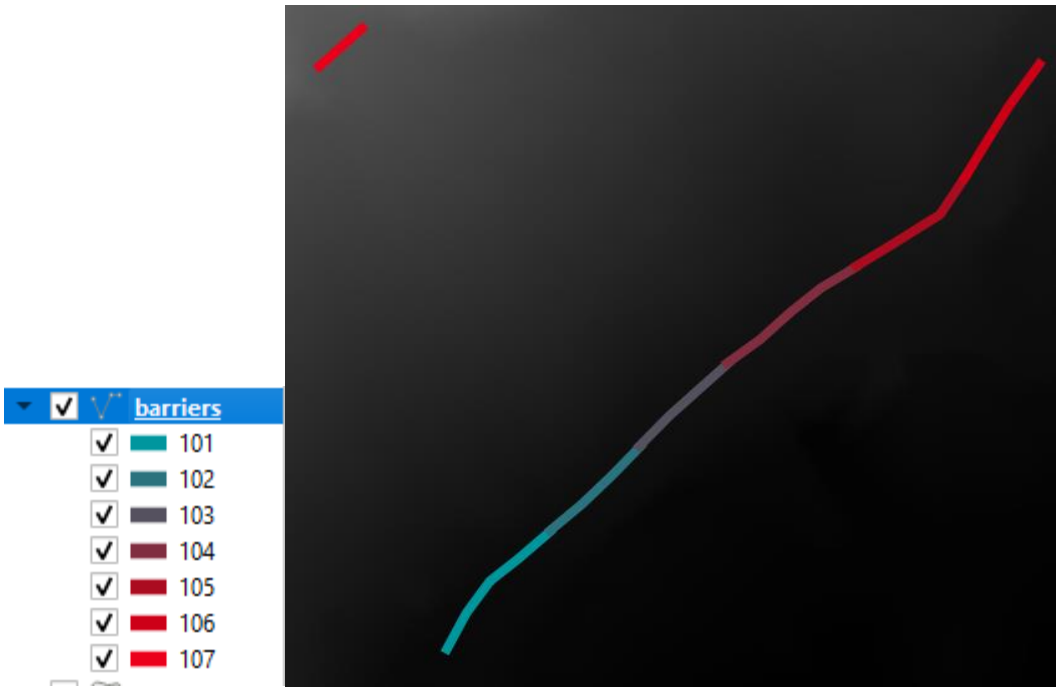

***Figura 4**. Información lineal correspondiente a los tramos de barrera existentes, a la derecha se muestra el identificador de cada tramo, que se representa con un color diferente, ya que actúan de modo independiente al recibir los impactos.*

Ambas informaciones están georreferenciadas en el sistema de referencia oficial ETRS89 y sistema de coordenadas proyección UTM zona 31N.

4.2 Datos Ráster

Los modelos digitales de elevaciones (MDE) se facilitan en formato ráster, en forma de malla, donde cada celda representa la altura del terreno (cota ortométrica H), se puede entender esta asociación como una propiedad descriptiva, y la georreferenciación de la malla como su propiedad geométrica.

Para la zona de estudio que se presenta, el MDE (**Figura 5**) representa una ladera perteneciente al principado de Andorra, su estructura está formada por 555 columnas y 390 filas de celdas cuadradas de un metro de resolución, con una altura mínima de 975 m y una altura máxima de 1306 m. Esta georreferenciado con el sistema de referencia ETRS89, en proyección UTM 31N.

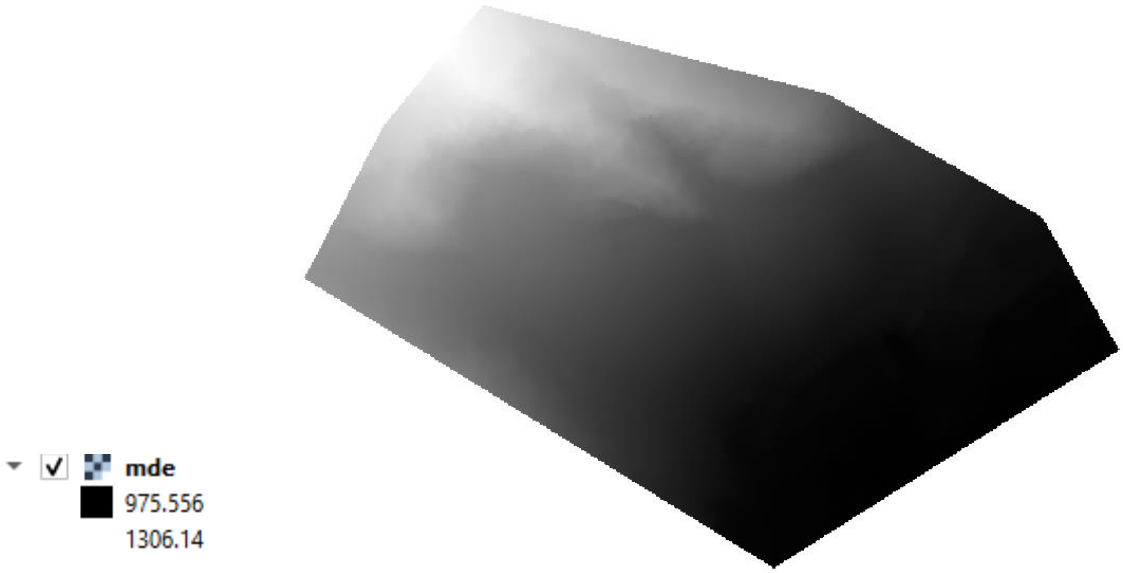

***Figura 5**. Modelo Digital de Elevaciones, muestra una zona del principado de Andorra, a la derecha se muestra la leyenda en una escalad e grises la cual representa las alturas.*

5 METODOLOGIA

La función principal del plugin, es la obtención de una sola capa ráster que contenga la información de los usos del suelo y las barreras, dando prioridad a estas últimas. Esta capa deberá tener las mismas características que el ráster que contiene el MDE:

Como se ha mencionado anteriormente, el método de trabajo consiste en que el usuario inserte un MDE representativo de la zona de estudio, la cual será habitualmente zonas montañosas susceptibles de sufrir desprendimientos, e información referente a tipos de suelo y barreras. Todas las capas han de estar en la misma proyección y sistema de coordenadas, para que se pueda realizar correctamente la combinación entre los datos.

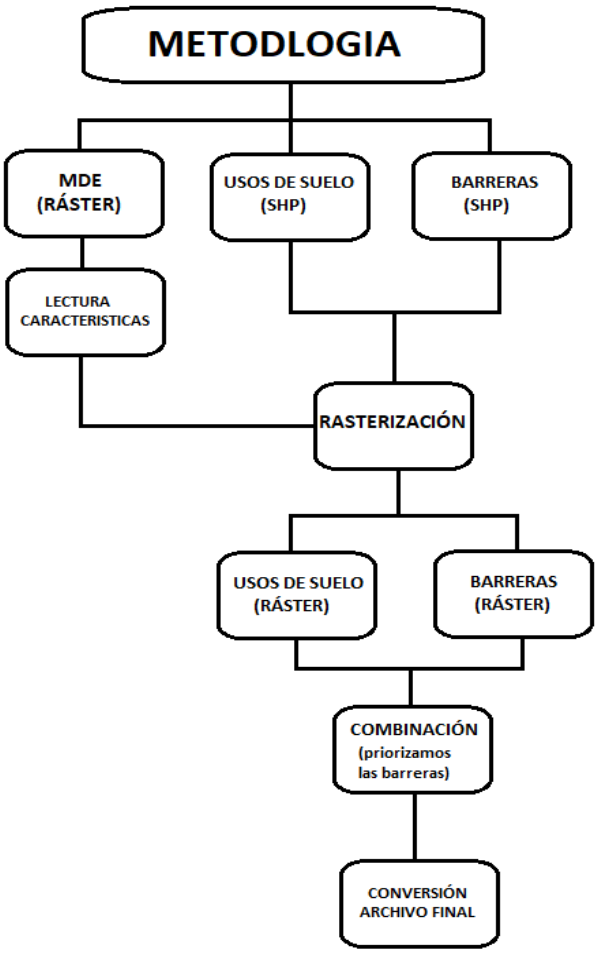

***Figura 6**. Flujo de metodología de trabajo*

El código en general se recoge en funciones que se encargan de realizar los procesos, es una manera de tenerlo organizado y ordenado. La **Figura 6** recoge el flujo de trabajo que se describe en detalle en las siguientes secciones.

5.1 Rasterización de datos

El primer paso al que se someten los datos, es la rasterización de los archivos que tienen un formato vectorial.

Es importante mencionar que se utilizan algoritmos pertenecientes a la biblioteca de GDAL. GDAL es una biblioteca de software para la lectura y escritura de formatos de datos geoespaciales, publicada bajo la MIT License por la fundación de código abierto (Open Source Geospatial Foundation). Como biblioteca viene con una variedad de utilidades en línea de comando para la traducción y el proceso de datos geoespaciales. Es importante recalcar que QGIS permite manejar formatos ráster y vectoriales a través de esta biblioteca, así como algunas bases de datos.

Después de conocer las posibilidades que nos ofrece esta biblioteca podemos buscar algún algoritmo dentro de este que nos permita conseguir el objetivo deseado. Para realizar pruebas de código es recomendable realizarlas a través de la IDLE propia de QGIS, ya que nos ofrece la posibilidad de comprobar la acciones que realiza el código in situ.

Antes de empezar a trabajar con la biblioteca debemos importar su módulo, para acceder a las funciones de procesamiento que nos ofrece, necesitamos realizar un '**import processing**', una vez realizado la importación de la biblioteca podemos acceder a los procesos que nos ofrece. [7]

Con la instrucción **QgsApplication.processingRegistry.algorithms()**, nos muestra una lista con todos los procesos que contiene (**Figura 7**), aquí también podemos ver el funcionamiento de cada algoritmo y ver que parámetros se necesitan para ejecutar el algoritmo.

Figura 7. *Listado de los algoritmos*

Dentro de la lista encontramos un algoritmo que se ajusta a nuestras necesidades, es un algoritmo que se llama '*rasterize*', según su definición, convierte geometrías vectoriales (puntos, líneas y polígonos) a imágenes ráster. Podemos encontrar más información en la documentación propias de QGIS. [3]

Processing.run ('gdal:rasterize' , INPUT, FIELD, UNITS, WIDTH, HEIGHT, EXTENSION, NODATA, DATA_TYPE,INVERT, OUTPUT)

Para poder utilizar el algoritmo son necesarios algunos parámetros:

- INPUT: Archivo vectorial que se desea convertir a imagen ráster.
- FIELD: Es el nombre del campo, cuyos valores son los utilizados en el algoritmo para la rasterización.
- UNITS: Unidades tamaño del ráster de salida.
- WIDTH: Establece la resolución horizontal del ráster de salida.
- HEIGHT: Establece la resolución vertical del ráster de salida.
- EXTENSION: Establece la extensión del ráster de salida.
- NODATA: Establece el valor de los pixeles que no tienen información.
- DATA_TYPE: Define el tipo de la imagen ráster resultante.
- INVERT: Invertir rasterización.
- OUTPUT: Nombre asignado al ráster de salida.

Una vez sabemos que es cada parámetro, debemos escribir las líneas de código necesarias para ejecutar el proceso. Para realizar la prueba es necesario insertar de manera manual los datos de entrada, que se guardan en variables, escribiendo el directorio y el nombre del fichero de entrada.

Para la rasterización de los datos vectoriales, se extraen del archivo ráster, en este case el MDE, los parámetros como pueden ser la extensión, la resolución vertical y horizontal, etc. Una vez tenemos dichos parámetro se pasan al algoritmo y este se ejecuta, convirtiendo un dato vectorial en una imagen ráster que tendrá las mismas características que el ráster del cual se extrajeron los parámetros.

Se ejecuta el algoritmo para ambos archivos, los cuales se quiere combinar la información. Respecto al código, para ejecutar los algoritmos en cada archivo de entrada se crean dos funciones, una para cada archivo, las funciones son ***def SoiltypeR***, ***def barriersR***. [Anejo I]

Una vez tenemos rasterizados los archivos el siguiente paso es la combinación entre ambos.

5.2 Información Combinada

Una vez tenemos los datos preparados, el siguiente paso es la combinación de información entre ambos, para ello recurrimos a otra de las herramientas incorporadas en QGIS, que sirve para realizar operaciones entre imágenes.

La calculadora ráster es una herramienta que permite realizar operaciones matemáticas sobre los valores de los pixeles existentes en un ráster, lo que es útil para la conversión y manipulación de dichos datos. El resultado de dichas operaciones se muestra en una capa ráster y en un formato admitido por la librería GDAL.

Permite el uso del '*raster calculator*' desde la consola de python a través de las clases QgsRasterCalculator y QgsRasterCalculatorEntry del módulo qgis.analysis de PyQgis.

QgsRasterCalculatorEntry()

Primeramente, creamos objetos de la clase QgsCalculatorEntry, que sirve para guardar la información de los ráster, creamos dos objetos cada uno con la información del ráster que se quiere interactuar en la calculadora ráster. Para ello creamos una lista e introducimos en la lista ambos objetos, para luego pasarlo como parámetro en la calculadora ráster.

QgsRasterCalculator (FORMULA 'String', OUTPUTFILE, OUTPUTFORMAT, OUTPUTEXTENT, OUTPUTCOLUMNS, OUTPUTROWS, RASTERENTRIES)

- FORMULA: En este apartado se introduce la formula con la que se quiere combinar ambos ráster. Ha de ser en formato String.
- OUTPUTFILE: Ruta del directorio y nombre del archivo de salida.
- OUTPUTFORMAT: Define el tipo de la imagen ráster resultante.
- OUTPUTEXTENT: Establece la extensión del ráster de salida.
- OUTPUTCOLUMNS/WIDTH: Establece la resolución horizontal del ráster de salida.
- OUTPUTROWS/HEIGHT: Establece la resolución vertical del ráster de salida.
- RASTERENTRIES: Objetos de tipo QgsRasterEntry con la información de los ráster a combinar.

Una de las partes más importantes en este punto es la expresión de la manera en que las imágenes se combinan. Para este proyecto se ha establecido que el ráster con información de las barreras tiene prioridad sobre el ráster con información de los suelos. La fórmula empleada es la siguiente:

***Figura 8**. Ráster de Usos de Suelo (R1)*

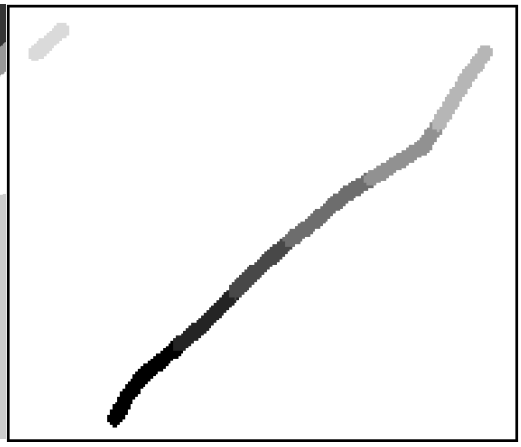

***Figura 9**. Ráster de Barreras (R2)*

$$((R2 > 0) * R2) + ((R2 = 0) * R1)$$

***Ecuación 1**.Ecuación de combinación ráster*

La fórmula (**Ecuación 1**) indica que los valores que son igual a 0 en la imagen con información de barreras R2 (**Figura 9**), se sustituyan por la información de la imagen con información de tipos de suelo R1 (**Figura 8**), y los valores que contienen información se mantienen. Es decir, la información de la barrera prevalece sobre la de uso del suelo.

Haciendo referencia al código, se observa [Anejo I], la función ***def F3***, es la encargada de ejecutar el algoritmo, en la función se pasan muchos parámetros, ya que necesita acceder a todos los archivos, tanto vectoriales como rásters, para recoger la información de estos. Bien una vez se tiene claro cómo se quieren combinar las imágenes únicamente se completa el algoritmo con los parámetros que faltan y se ejecuta.

Hasta este punto del proyecto, se obtiene el ráster con información combinada, uno de los principales objetivos del proyecto. Únicamente queda realizar una conversión del archivo, ya que al ejecutar la función el archivo de salida adopta una extensión llamada GeoTiff.

GeoTiff es un estándar de metadatos, permite que información georreferenciada sea encajada en un archivo de formato TIFF.

Convertir el archivo a un formato más común y utilizado, lo hará más manejable y sencillo de compartir, el formato ASCII es el más adecuado.

5.3 Conversión del archivo resultante

Una vez tenemos el archivo creado correctamente, realizamos una conversión de formato, es necesario ya que el formato de lectura utilizado en las simulaciones de desprendimiento rocosos ha de ser un formato ASCII, también es una forma de mantener el archivo en un formato compatible con otros programas y reducimos el tamaño del mismo.

Para ello recurrimos de nuevo a los procesos que incluye GDAL, dentro de la lista de algoritmos existe uno adecuado al problema, el algoritmo es *'gdal:translate'*, según su definición se puede usar para convertir datos ráster entre diferentes formatos, lo que podría realizar algunas operaciones como subconjuntos, remuestreo y cambio de escala de pixeles en el proceso.

Como es común en el código existe una función para ejecutar el algoritmo, ***def Convert***.

Processing.run ('gdal:translate', INPUT, TARGET_CRS, NODATA, COPY_SUBDATASETS, DATA_TYPE, OUTPUT)

- INPUT: Archivo de entrada el cual se quiere cambiar el formato.
- TARGET_CRS: Establecer la proyección para el archivo de salida.
- NODATA: Establecer el valor para los pixeles que no tienen información.
- COPY_SUBDATASETS: Copiar todos los subgrupos de datos del archivo de entrada al archivo de salida.
- DATA_TYPE: Define el tipo de la imagen ráster resultante.
- OUTPUT: Nombre de la imagen resultante.

Una vez ejecutado el último paso obtenemos un archivo en formato ASCII, podemos abrir el archivo con un simple bloc de notas para ver que contiene y si tiene la lógica resultante de un ráster con formato ASCII.

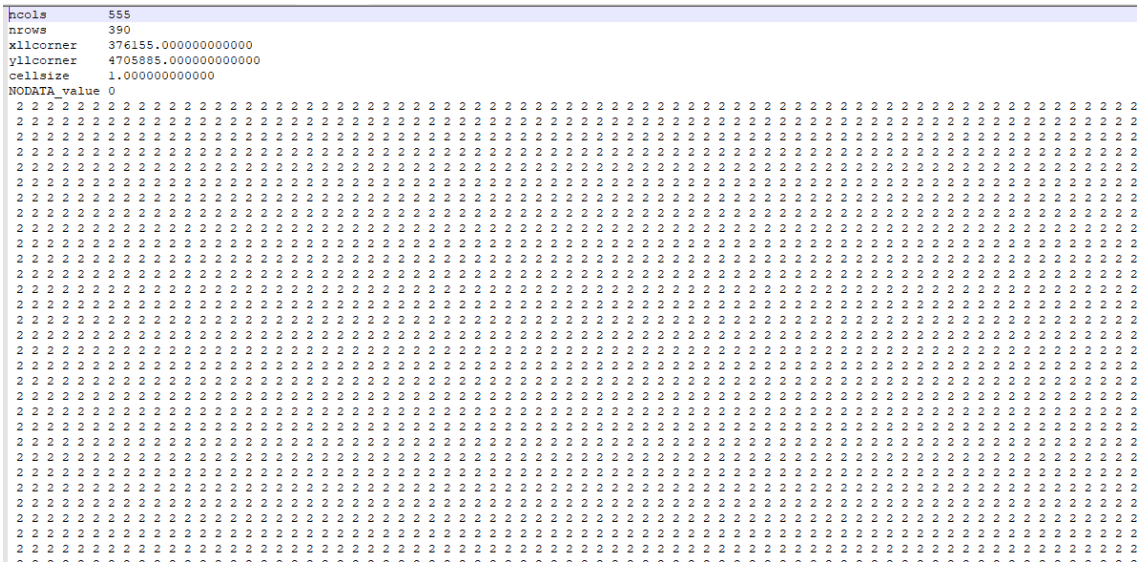

***Figura 10**. Muestra la estructura que ha de tener el archivo para su funcionalidad.*

Como podemos ver (**Figura 10**), el archivo tiene un encabezado con la información del ráster, número de filas y columnas, tamaño del pixel, extensión, etc. También contiene el valor de los pixeles en forma de matriz, por lo cual la conversión se ha realizado correctamente.

Llegados a este punto ya tenemos el archivo necesario para poder ejecutar las simulaciones con él, ahora el siguiente paso es automatizar el proceso para que el usuario pueda crear archivos de las mismas características, pero con diferentes datos de entrada. Para ello lo más habitual es crear un plugin, herramienta que se utiliza en QGIS para realizar diferentes procesos.

5.4 Creación del complemento

5.4.1 Plugin Builder

Para facilitar el trabajo a la hora de crear el plugin, existe una herramienta capaz de crear la estructura básica que lo compone, Plugin Builder es un plugin creador de plugins.

Una vez se ejecuta, hay que seguir diferentes pasos donde se especifica el tipo de plugin y varios datos necesarios, que se explican a continuación (**Figura 11**).

- Class name: Es el nombre que utilizara el complemento (Plugin Builder), para crear la clase en Python, el nombre tiene que ser en CamelCase (estilo de escritura que se aplica a frases o palabras compuestas, intercala mayúsculas y minúsculas sin espacios).

- Plugin name: Es el nombre del plugin que se creará, es el nombre que aparece en el gestor de plugins, y el nombre que aparece en el menú/toolbar en QGIS.

- Description: Breve descripción de la funcionalidad del plugin, se muestra en el gestor de complementos, se pueden añadir fotos o enlaces del plugin.

- Module name: Es el nombre que será utilizado para crear el módulo Python, es el archivo que se crea junto con el complemento, tiene que estar escrito en minúsculas y las palabras separadas por guiones bajos.

- Version number: es el número de versión del complemento. Qgis utiliza este parámetro para identificar los complementos y actualizarlos en caso de haber nuevas versiones, o versiones actualizadas.

- Minimum QGIS version: Versión mínima de Qgis en la que el complemento puede funcionar, esto es importante, ya que con la nueva versión de Qgis, ha habido muchos cambios referentes al motor interno de Qgis.

- Author/Company: Nombre o entidad a la que pertenece el desarrollo del plugin. Esta información se utiliza para declarar los derechos de autor y propiedad intelectual de los archivos origen.

- Email adress: Dirección de correo electrónico del desarrollador o de la entidad desarrolladora, sirve para la comunicación con el desarrollador, por parte de los usuarios o por la organización QGIS.

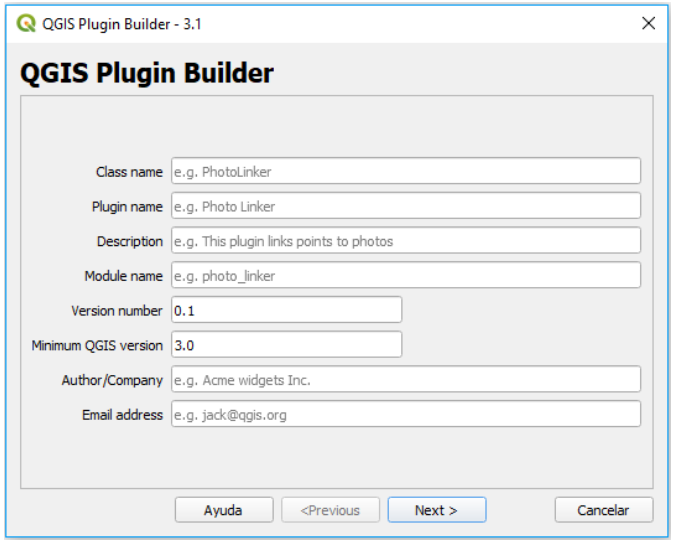

Figura 11. *Ventana de Plugin Builder para introducir información referente al plugin*

La siguiente pestaña que se muestra, sirve para dar una descripción más detallada y si es necesario alguna instrucción para facilitar el uso del plugin.

Seguidamente (**Figura 12**), se escoge el tipo de plantilla que adoptara el plugin, un texto para el menú y en que menú queremos que aparezca el plugin, Qgis tiene varios menús según el tipo de datos que se están manejando, este se colocará en el menú que mejor se ajuste a su aplicación.

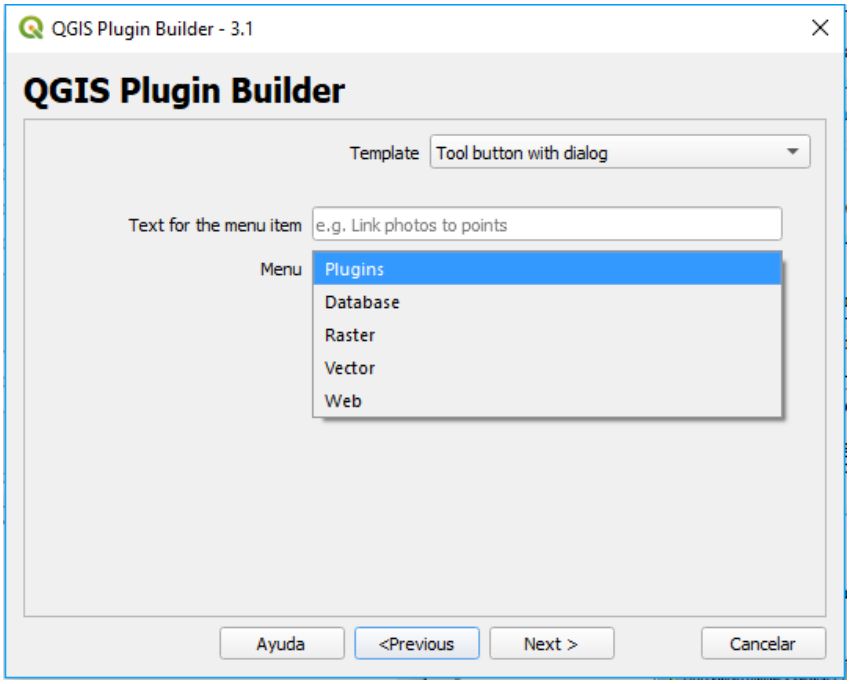

Figura 12. *Ventana de Plugin Builder para seleccionar la plantilla y escoger el menú de herramientas donde aparecerá el plugin.*

En el penúltimo paso (**Figura 13**), en la ventana se muestran unos campos que se requieren para facilitar la mejora y errores que pueda contener el plugin.

- Bug tracker: Se pide una URL donde queden registrados los errores del plugin y puedan tener seguimiento. Si el código es extensoy profesional lo más eficaz es crear un proyecto en http://github.com o utilizar por defecto http://hub.qgis.org/projects/new.
- Repository: Requiere la URL del repositorio del código fuente del plugin. Esto permite que cualquier usuario pueda modificar o adaptar el código.
- Home page: URL de la página principal del complemento, puede ser la misma que la página del proyecto, a Github.
- Tags: Son las etiquetas del complemento, las etiquetas son una lista de palabras claves, para describir o encontrar el plugin.
- Por ultimo nos da la opción de marcar el complemento como experimental, sirve para indicar que el complemento está en prueba o que está incompleto, y puede tener comportamientos no esperados.

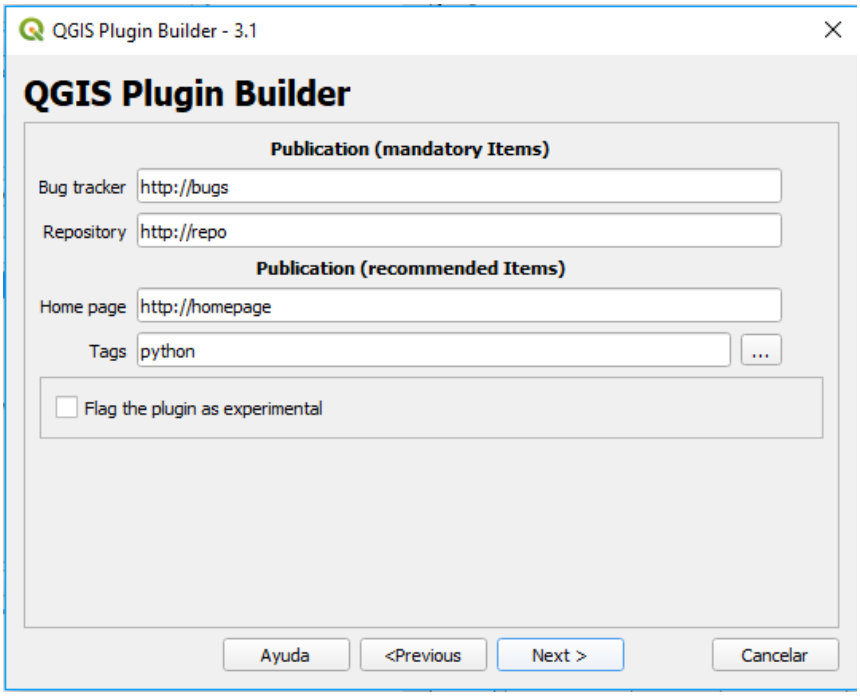

***Figura 13**. Ventana de Plugin Builder para la introduccion de directorios para la publicacion y el seguimiento.*

El último paso, antes de generar el plugin, es indicar la ruta donde queremos que se cree. Esta ruta no tiene gran importancia, pues una vez este operativo debemos incluirlo a QGIS colocando el plugin en su ubicación correspondiente.

5.4.2 Plugin Reloader

Plugin Reloader es un plugin experimental incorporado en QGIS, es muy útil a la hora de realizar cambios en el código, pues nos permite recargar un plugin dentro de QGIS, y permite actualizarlo, de tal forma que se ven las modificaciones que se van realizando. (**Figura 14**)

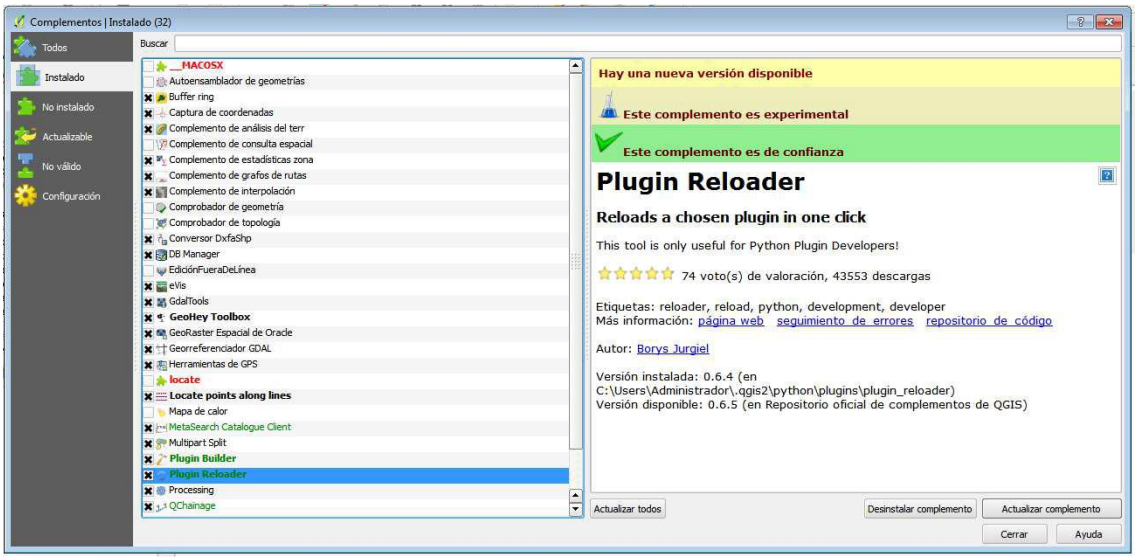

***Figura 14**. Instalación de Plugin Reloader*

5.4.3 Instalación y preparación del complemento

Como se ha mencionado anteriormente, el Plugin Builder crea la estructura básica. La estructura mínima de archivos que debe contener un plugin son:

- **__init__.py**
- **icon.png**
- **metadata.txt**
- **resources.qrc**
- **resources.py**
- **primer_plugin.py**
- **primer_plugin_dialog.py**
- **primer_plugin_dialog_base.ui**

__init__.py es el punto donde se inicia el plugin. Es obligatorio para el sistema de importación de Python. Debe contener, al menos, un método llamado classFactory con el iface como argumento.

icon.png Icono que se muestra en la barra de herramientas. El icono debe tener 24x24 pixels en formato PNG.

metadata.txt contiene información sobre el plugin: versión, autor, email, la descripción, la versión, el ícono y la versión mínima de QGIS. Este archivo es necesario para que QGIS reconozca el complemento.

primer_plugin.py La implementación principal de su complemento que maneja la carga, descarga y ejecución de las funciones del complemento.

Es el nombre que le hemos dado a nuestro plugin. Contiene el script con el código principal. Las acciones principales se recogen en él.

Como mínimo, dentro de este archivo deben figurar las siguientes funciones:

- __init__: que da acceso a la interfaz de QGIS
- initGui (): para iniciar el plugin
- unload (): para finalizar la acción del plugin

resources.qrc Describe los recursos (por ejemplo, icon.png), utilizados por el complemento y los formularios GUI.

resources.py es el archivo Python generado desde resources.qrc por el compilador pyrcc5.

primerplugin_dialog.py El archivo de Python generado a partir de ui_primerplugin.ui por el compilador de interfaz de PyQt, pyuic5.

NombrePlugin_dialog_base.ui El archivo de interfaz GUI creado por Qt Designer.

Ahora debemos compilar el archivo resources.qrc para que Python lo pueda entender, es decir, necesitamos traducir los tipos de archivo '.qrc' a archivos de tipo '*.py'
Desde la terminal de OsGeo4WShell, que viene incorporado en el paquete de instalación de QGIS OsGeo, una vez abierta la terminal accedemos al directorio donde tenemos nuestro complemento:

cd C:\Users\USUARIO\Desktop\TFG\Datos\Plugin final\rockterreno

Antes de ejecutar el compilador necesitamos llamar a los archivos .bat que contienen las rutas de inicialización para poder acceder a las librerías y archivos necesarios para la compilación de Qt5 y Python3, de la siguiente forma:

call qt5_env.bat
call py3_env.bat

Ahora ya se puede ejecutar el compilador y transformar el archivo '.qrc' a '.py' (**Figura 15**):

pyrcc5 –o resources.py resources.qrc

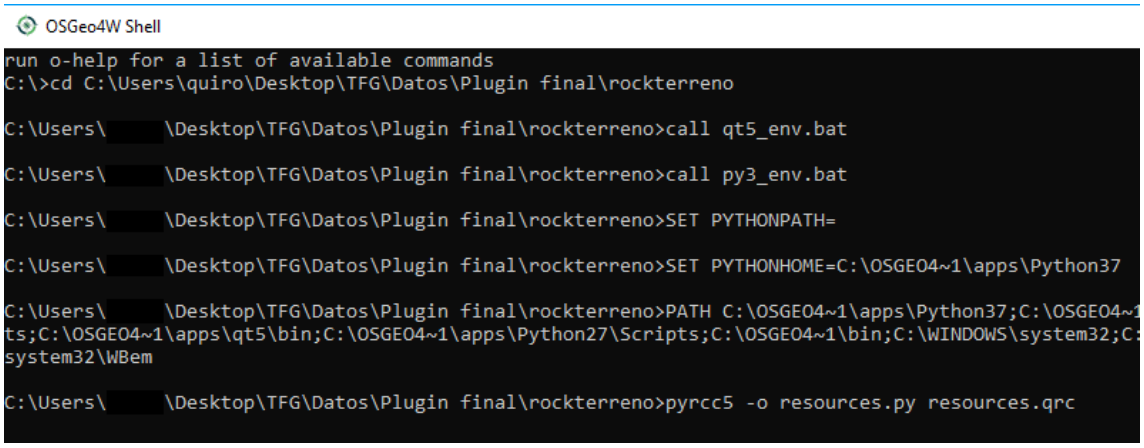

***Figura 15**. Compilación del complemento*

Llegados a este punto ya tenemos el complemento para que QGIS pueda leerlo, para instalarlo dentro de QGIS debemos dirigirnos al gestor de complementos y activar el plugin que se ha creado. Por ultimo falta añadir la funcionalidad con nuestro código y crear una interfaz gráfica donde resulte más fácil interactuar al usuario con él.

5.4.4 Interfaz Gráfica

Para comodidad del usuario se crea una interfaz donde interactuar con la aplicación sea sencilla y entendible, para ello requerimos acceder a QtDesigner, es una aplicación que viene incorporada en la instalación de QGIS OsGeo.

Dentro de la estructura de archivos que se ha creado, el archivo **NombrePlugin_dialog_base.ui** es el que contiene la interfaz de usuario, desde la aplicación se busca y se abre dicho archivo. (**Figura 16**)

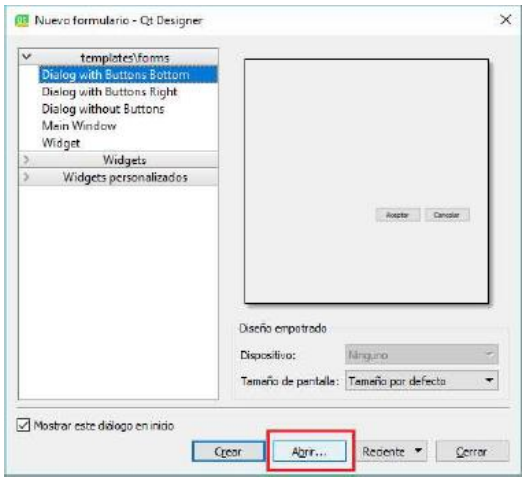

***Figura 16**. Ventana QtDesigner para abrir el archivo .ui*

El archivo tal y como se crea únicamente tiene un formulario con dos botones (**Figura 17**), a continuación, se pueden añadir objetos para que el usuario pueda ejecutar el complemento y definir los parámetros e insertar los datos necesarios para su ejecución.

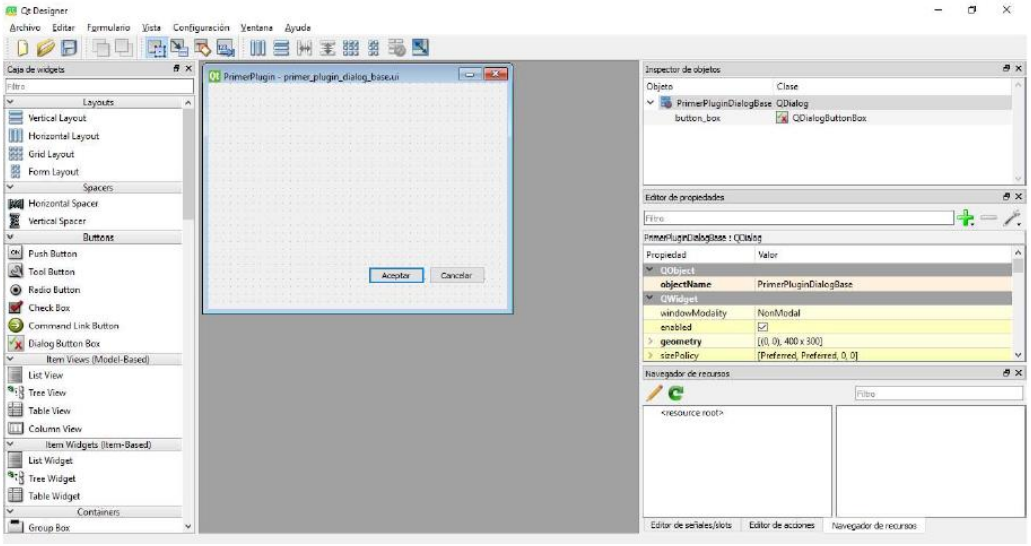

Figura 17. *Interfaz de QtDesigner, con la ventana del archivo .ui vacío, a la derecha se muestra el listado de objetos y a la derecha la información relativa a estos.*

En la imagen (**Figura 18**), se muestra el programa QtDesigner. A la izquierda se muestra el listado de herramientas, botones y widgets del cual se seleccionan y se arrastran al formulario.

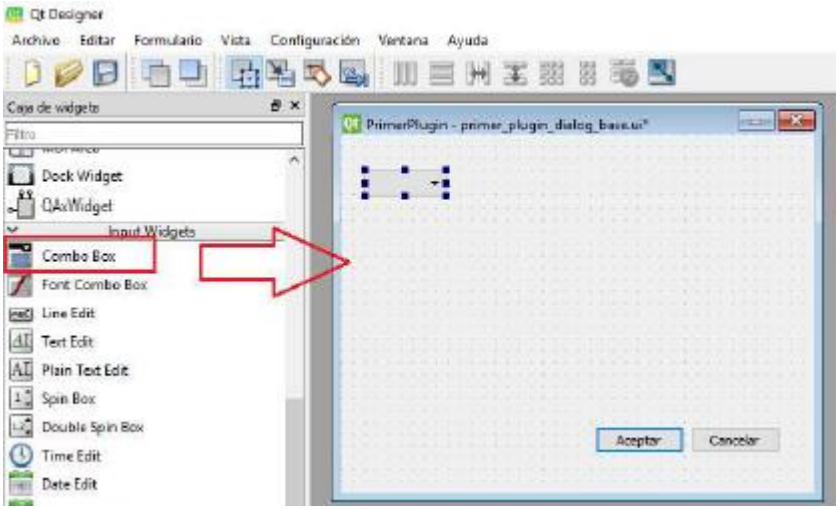

Figura 18. *Inserción del objeto ComboBox al archivo .ui*

Para ejecutar el complemento son necesarios varios objetos y widgets que se describen a continuación:

- **comboBox:** Es una caja desplegable que muestra los elementos en una lista para escoger la capa a utilizar. Se utiliza para que los elementos que se muestran son aquellos que aparecen en la TOC del programa QGIS, el usuario escoge una como capa de entrada para el proceso.

- **lineEdit:** Es una caja donde permite al usuario escribir una cadena de caracteres, como nombres rutas de directorios, etc. Este objeto sirve para acceder a las carpetas donde se selecciona una capa de entrada o donde se selecciona el directorio donde se genera el archivo final.

- **pushButton:** Es un botón que da acceso a la estructura interna del sistema de cada usuario. Se conecta con el objeto lineEdit, la ruta que se selecciona en pushbutton, se aplica directamente en lineEdit.

- **spinBox:** Es un objeto con dos botones (uno para añadir y otro para disminuir), solo acepta cifras numéricas. En este caso se utiliza para crear un buffer si fuera necesario.

- **checkBox:** Es una caja donde te permite activar o desactivar el objeto, cuando el objeto está activo se muestra con una palomita en la caja. Se utiliza para borrar o mantener los archivos intermedios generados durante el proceso, por si el usuario necesitara acceder a ellos.

- **labels:** Son etiquetas, tienen la función de mostrar cadenas de texto para identificar los objetos de la interface.

Cada objeto creado mediante QtDesigner tiene su propio nombre (**Figura 19**), se puede modificar, pero no se pueden repetir, pues luego en el código se hace referencia a cada objeto a través de su nombre

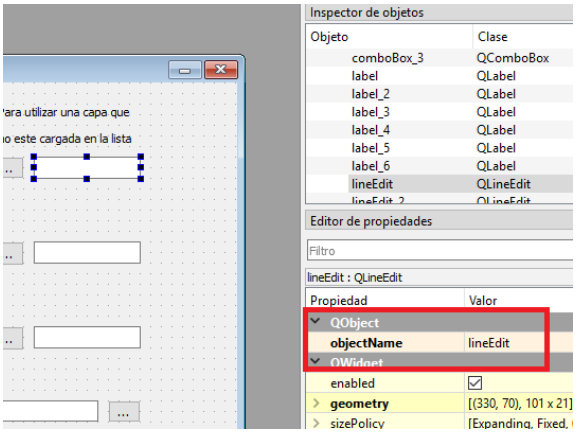

***Figura 19**. Nombre del objeto insertado, en el caso un lineEdit*

Para crear el diseño del complemento solo necesitamos añadir cada objeto que se precisa entre la lista de objetos y recursos que ofrece el programa, e ir introduciéndolos en la interfaz del complemento, con la configuración deseada, finalmente se guardan los cambios. En la siguiente imagen se muestra el resultado final de la interfaz gráfica: (**Figura 20**)

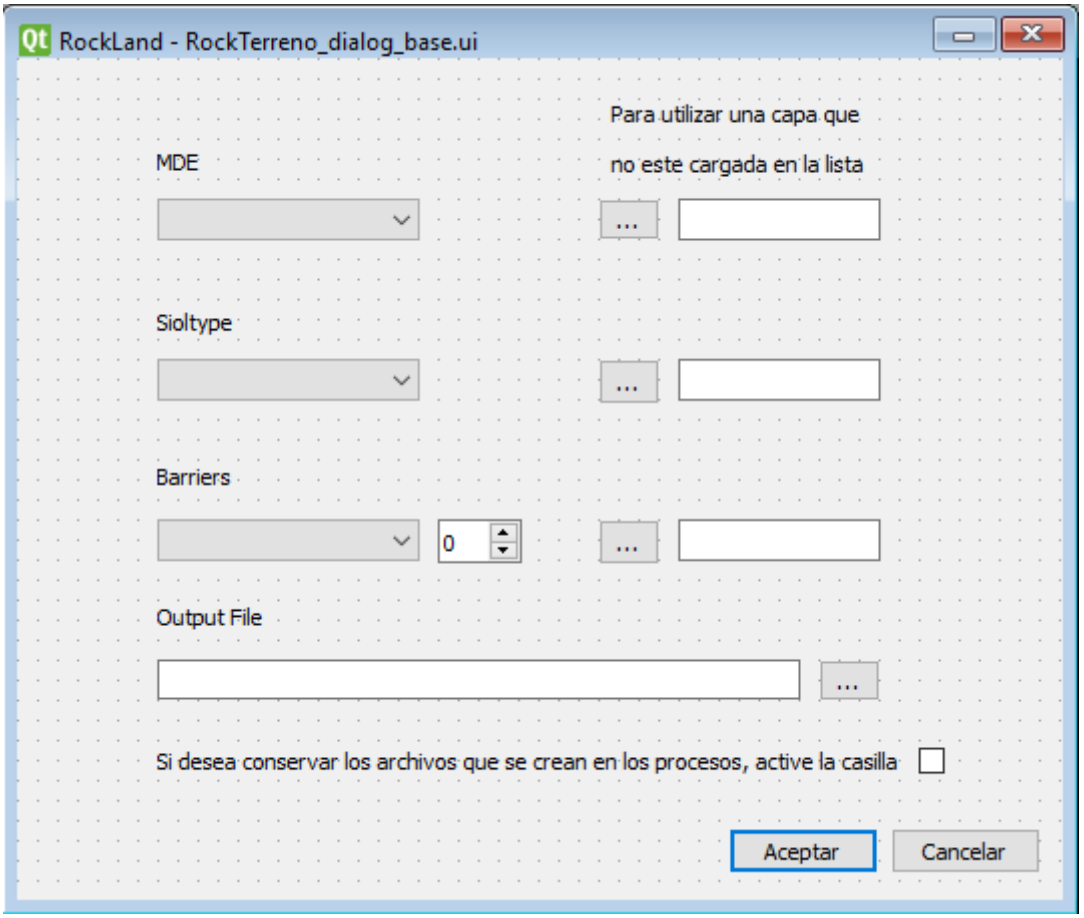

Figura 20. *Interfaz Gráfica con los objetos insertados, aspecto final del plugin.*

5.4.5 Funcionalidad del complemento

Una vez tenemos la interfaz gráfica creada, y sabemos los elementos que la componen, tenemos que conectarlos, de manera que interactúen de la forma que deseamos, para ello se requiere acceder al código y desde este enlazaremos los métodos con los objetos. Accediendo al archivo con extensión .py (extensión de Python), dentro de la carpeta que compone el plugin, desde la IDLE Sublime Text (apartado 3.4).

Hay mucha parte del código creada automáticamente al utilizar el complemento Plugin Builder (apartado 5.4.1), se añaden las líneas de código que enlazan con los objetos de la interfaz gráfica.

Tienen una estructura específica, si nos fijamos en la primera parte del código generada automáticamente, se observa la siguiente función ***def run(self)***, junto con el comentario:

"""Run method that performs all the real work"""

Esto indica que hasta esta parte son líneas que componen el plugin sin funcionalidad, es decir, son líneas que pertenecen a la inicialización del mismo y la conectividad a QGIS, como puede ser la visualización su icono, hasta su desinstalación del mismo, para que no quede rastro alguno y no haga intromisión con otros plugins (son las figuras básicas que debe contener, apartado 5.1.2).

A partir de este punto es donde entramos en el código que especifica la funcionalidad del plugin insertando, las líneas de código propio. La estructura que se representa en el código consta de la declaración de las funciones creadas anteriormente, como pueden ser ***def SoiltypeR***, ***def barriersR***, ***def F3***, estas son las tres funciones básicas que componen el código, juntamente con otras complementarias, estas funciones se declaran antes de la función que hace correr el código ***def run(self)***. [9,10,11]

Dentro de la función ***def run(self)***, se ejecuta la llamada a las demás funciones, se realiza de esta manera para tener clara la estructura y poder ver que hace cada parte del código.

Primeramente, se detalla la forma de conectar los objetos de la interfaz al código, lo primero que interesa es mostrar las capas que el usuario tiene cargada en la TOC de QGIS, y las muestre en forma de lista, donde este puede seleccionar la capa que convenga. Para ello utilizamos el widget *comboBox*, este es una clase perteneciente a la biblioteca PyQt5, como mencionamos antes es un binding que nos permite comunicarnos desde Python. Se utilizan las siguientes instrucciones, para que el código lea las capas y las añade a una lista desplegable en el objeto comboBox.

Nota: Qwidget es la clase base de todos los objetos de la interfaz de usuario

```
layers=qgis.core.QgsProject.instance().layerTreeRoot().children()
    listR=[layer.name() for layer in layers if layer.layer().type()==QgsMapLayer.RasterLayer]
    self.dlg.comboBox.addItems(listR)
    #QgsProject.instance().layerTreeRoot.insertLayer(0,listR)
    listV=[layer.name() for layer in layers if layer.layer().type()!=QgsMapLayer.RasterLayer]
    self.dlg.comboBox_2.addItems(listV)
    self.dlg.comboBox_3.addItems(listV)
```

De esta forma creamos una lista y con el método '*.addItems()*' añadimos las capas a la lista, sería suficiente si de manera iterativa cada vez que se ejecuta el código se añaden las mismas capas, apareciendo duplicadas en cada ejecución, por tal de mantener el plugin limpio y claro se ejecuta una sentencia que elimina las capas añadidas anteriormente.

```
#Vacia el comboBox ya que se acumulan las entradas
while self.dlg.comboBox.count() > 0:
    self.dlg.comboBox.removeItem(0)
```

En las sentencia se cuentan los elementos que hay en la lista con el método '*.count()*', si existe alguna capa en la lista de la anterior ejecución, se eliminan con el método '*.removeItem()*'.

Para ir comprobando las modificaciones que realizamos, guardamos el archivo con las modificaciones realizadas en la IDLE, y recargamos el complemento desde el Plugin Reloader, y lo ejecutamos en QGIS, de esta forma podemos ver los cambios que vamos realizando in situ. Ahora podemos ver como al ejecutarlo se muestran las capas en el comboBox.

Para darle una mayor versatilidad al plugin se ha añadido la posibilidad de cargar una capa externa, es decir, una capa que no se encuentre cargada en la TOC de QGIS, para ello el usuario necesita acceder al sistema de ficheros de su equipo, ver su estructura, escoger el directorio y el fichero que desea añadir, de otra manera debería escribir y conocer la ruta entera donde se encuentra dicho fichero. Para esta funcionalidad necesitamos importar el módulo *QFileDialog*, junto a las demás librerías al principio del código. De esta forma se define la ruta junto con el nombre del fichero y se pasa como cadena de texto a *lineEdit* (otro objeto de la interfaz gráfica). Esta es una función independiente como las anteriormente mencionadas, pues ha de tener la misma característica y ha de esta declarada antes de la función ***def Run(self)***, y tiene la siguiente organización:

```
def select_input_file(self):
    inputfile, _filter=QFileDialog.getOpenFileName(self.dlg,"Select input file","","All Files (*);;Text Files (*.txt)")
    self.dlg.lineEdit.setText(str(inputfile))
```

Esta función une la ruta junto con el nombre, y la pasa como cadena al lineEdit con el método '*.setText()*', la siguiente instrucción:

self.dlg.pushButton.clicked.connect(self.select_input_file)

Conecta con el objeto QpushButton, de esta manera cuando clicamos al botón, se ejecuta la función **def *select_input_file(self)***, que es el que accede al sistema de archivos propio de cada usuario.

Por otra parte, creamos una nueva función que realiza el proceso inverso, guarda el fichero de salida y como en la función anterior, el usuario puede escoger el directorio y el nombre, utilizamos el mismo módulo *QFileDialog*, y se pasa de nuevo, en este caso, a un nuevo objeto *lineEdit4*, se mencionó anteriormente que cada objeto de la interfaz ha de tener su propio nombre y no puede estar repetido, aquí se puede ver una de las causas por las que se deben de tener clara la estructura de la interfaz gráfica.

```
def select_output_file(self):
    filename, _filter = QFileDialog.getSaveFileName(self.dlg,"Select output file ","","*.asc")
    self.dlg.lineEdit_4.setText(filename)
```

Ya tenemos las funciones necesarias, pero como en el *comboBox*, cada vez que se ejecuta el plugin este mantiene los elementos introducidos en la anterior ejecución, entonces *lineEdit* muestra la ruta y el nombre que se seleccionó en la última vez que fue utilizado, para mantener el plugin limpio y que no sea molesto y pesada su utilización, se ha de eliminar las ultimas entradas, para ello utilizamos las siguientes instrucciones:

```
self.dlg.lineEdit.clear()
self.dlg.lineEdit_4.clear()
```

Estas instrucciones se insertan en el apartado de la ejecución del código propio, dentro de la sección de la función ***def Run(self)***, nuevamente guardamos los cambios en la IDLE, se recarga el plugin en QGIS a través del complemento Plugin Reloader, se comprueban los cambios realizados y que todo funciona correctamente.

Hasta este punto el plugin es funcional, pero se añaden unos pequeños añadidos que posiblemente el usuario necesite usar, como puede ser modificar uno de los datos de entrada, realizando un buffer. Para ello utilizamos un objeto llamado *spinBox* (comentado en el apartado 5.1.3), se relaciona en el código como cualquier otro objeto, resulta bastante sencillo, únicamente se relaciona a una variable que recoge el número que se introduce dentro del *spinBox*, y se pasa como parámetro del proceso donde se necesita. Se relaciona de la siguiente forma:

```
tamañobuffer=self.dlg.spinBox.value()
```

Como pasa en todos los objetos del plugin, este recoge el último dato registrado en la última actuación, para ello establecemos que cada vez que se quiera utilizar el plugin este valor por defecto sea 0, para ello:

```
self.dlg.spinBox.setValue(0)
```

El ultimo objeto que se muestra en la interfaz es un *checkBox* (apartado 5.1.3), se ha establecido que se utiliza este objeto de manera que el usuario puede decidir guardar o eliminar, los datos intermedios que se generan en los diferentes procesos. Para ello se crea una función que se llama ***def remove()***, y se relaciona con el código con la siguiente sentencia:

```
borrafiles=self.dlg.checkBox.isChecked()
```

De manera que este detecta si el *checkBox* está activo, con el método '*isChecked()*', llama a la función y la ejecuta, de lo contrario, si el chekBox no está activo, no ejecuta la función y los datos intermedios no se eliminan, se utiliza esta alternativa, por si el usuario quiere trabajar con los dichos datos, si quiere comprobar que las ejecuciones se realizan correctamente o quiere saber cómo trabaja el plugin internamente.

Hasta este punto del proyecto ya se tiene una interfaz completamente conectada con el código y, para que el usuario pueda trabajar con ella, de tal manera que una vez realizados estos cambios y guardados, se recarga el plugin y se comprueban los cambios, en el *comboBox* aparecen las capas, el *pushButton* accede al sistema de ficheros etc. Todavía el plugin no se ejecuta correctamente ya que se han de pasar todos los datos que el usuario introduce en la interfaz, se han de pasar como variables dentro de los procesos, que son los que ejecutan l código correctamente.

Los cambios que han de realizarse son, cambiar la información de entrada para que sea la que se recoge en el *comboBox*, que la información de entrada y salida del proceso coja los datos introducidos en el *lineEdit*, en lugar de la ruta establecida previamente y por último que los parámetros numéricos recogidos en el *spinBox*, se sustituyan por el valor por defecto establecido anteriormente.

Finalmente podemos realizar las comprobaciones necesarias y actualizándolo, disponemos del plugin funcional instalado en QGIS, con las funciones necesarias y los valores correctos para la generación del archivo final de salida.

Haciendo un resumen de todos los pasos, primero creamos el código que tiene la funcionalidad, creamos la interfaz gráfica y luego la conectamos con el código para que recoger la información que introduce el usuario en la interfaz gráfica.

6 RESULTADOS

A continuación, se realiza una demostración visual del funcionamiento del complemento, y de cómo se interactúa con el mismo.

Primero se accede a QGIS y abrir un nuevo proyecto y se cargan las capas necesarias en la TOC, no cal que las capas estén en el mismo sistema de referencia, ya q el código está pensado para que al ejecutarse se pueda seleccionar el sistema de referencia que mejor se ajuste a la zona de estudio, y hace una transformación de todas las capas al sistema de referencia seleccionado. Como los datos con los que se ha desarrollado este proyecto pertenecen a una zona montañosa del principiado de Andorra se utiliza el sistema de referencia y de coordenadas ETRS89 / UTM zona 31. Los sistemas de referencia llevan asociados un código que los identifica SRID, el cual en la mayoría de software, entro otras codificaciones, se define mediante los códigos definidos por el EPSG (*European Petroleum Survey Group)* (http://www.epsg.org/). Según está codificación la proyección UTM zona 31 en el sistema de referencia ETRS98 / corresponde con el código 25831. Una vez preparado el proyecto, se puede ejecutar el plugin. (**Figura 21**)

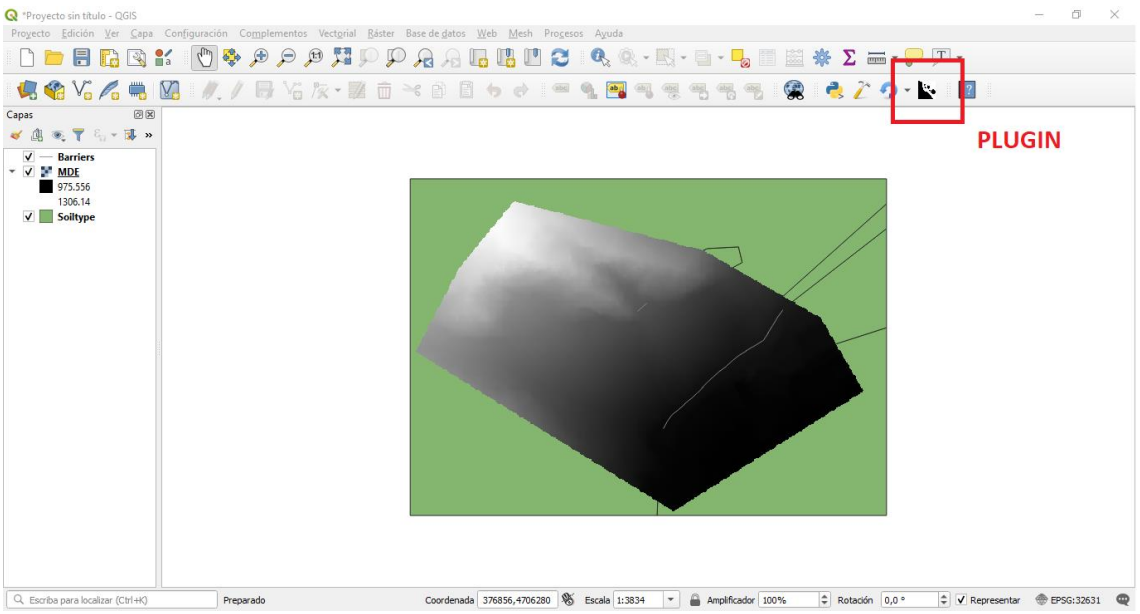

***Figura 21**. Interfaz de QGIS, donde se muestra el plugin*

Una vez ejecutado el plugin, aparece la ventana (**Figura 22**) con la interfaz gráfica creada en QtDesigner (apartado 5.4.3), se muestran todos los objetos introducidos (mencionados en apartado 5.4.3) con los que el usuario puede interactuar.

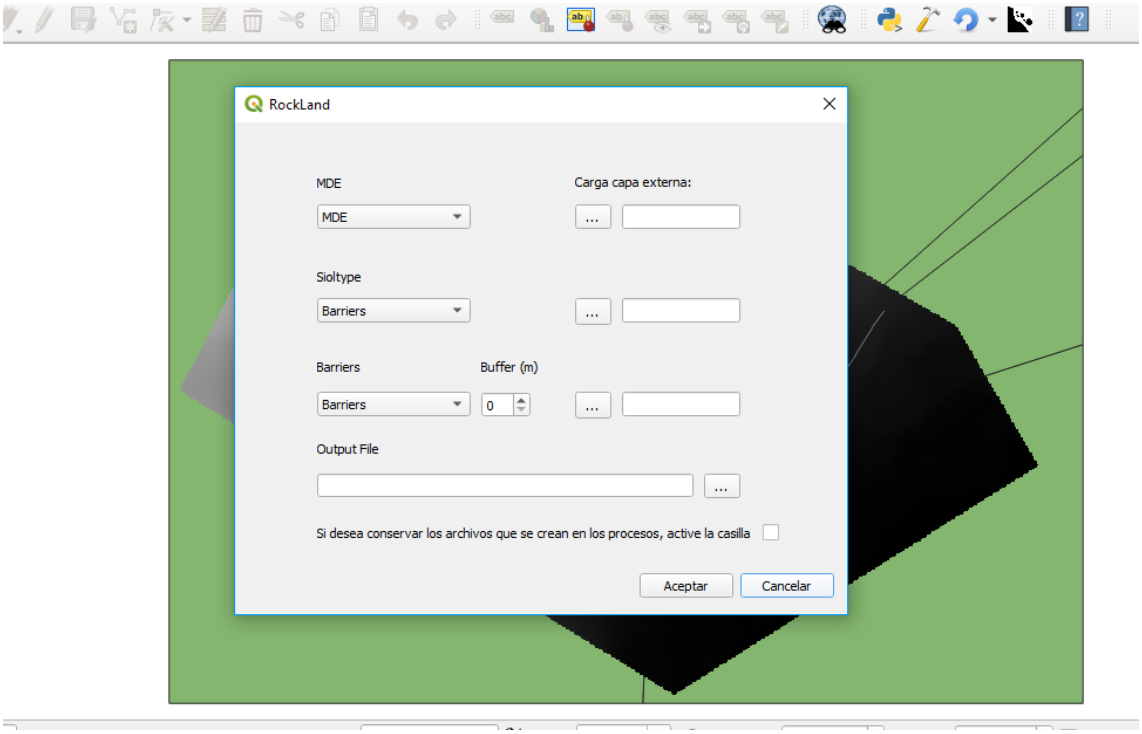

***Figura 22**. Interfaz Gráfica del plugin*

Como se puede comprobar en la **Figura 23**, cuando se despliega el *comboBox* (punto 1), aparecen las capas que se han cargado en la TOC, si se accede al *pushButton* (punto 2), se abre la ventana emergente que accede al sistema de archivos del propio usuario.

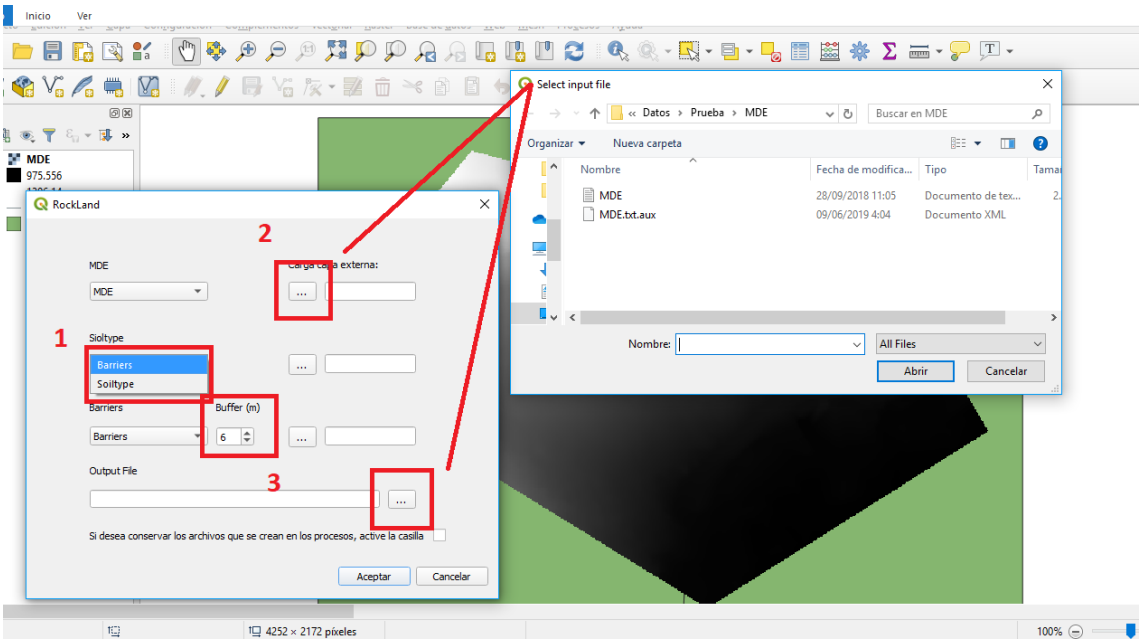

***Figura 23**. Funciones del plugin*

En estos apartados se pueden cargar capas externas que no se han cargado anteriormente en la TOC y se selecciona el directorio donde se guarda el archivo de salida. Cabe menciona que el *spinBox* (punto 3), es el objeto que coge el valor del buffer, en la **Figura 24** se puede ver la diferencia al añadir buffer a la capa.

***Figura 24**. Izquierda. Raster de Barreras con buffer aplicado; Derecha, Ráster de Barreras sin buffer aplicado*

En el siguiente ejemplo (**Figura 25**), se utiliza la opción de cargar capas externas (Punto 4) y se activa el *checkBox* (Punto 5).

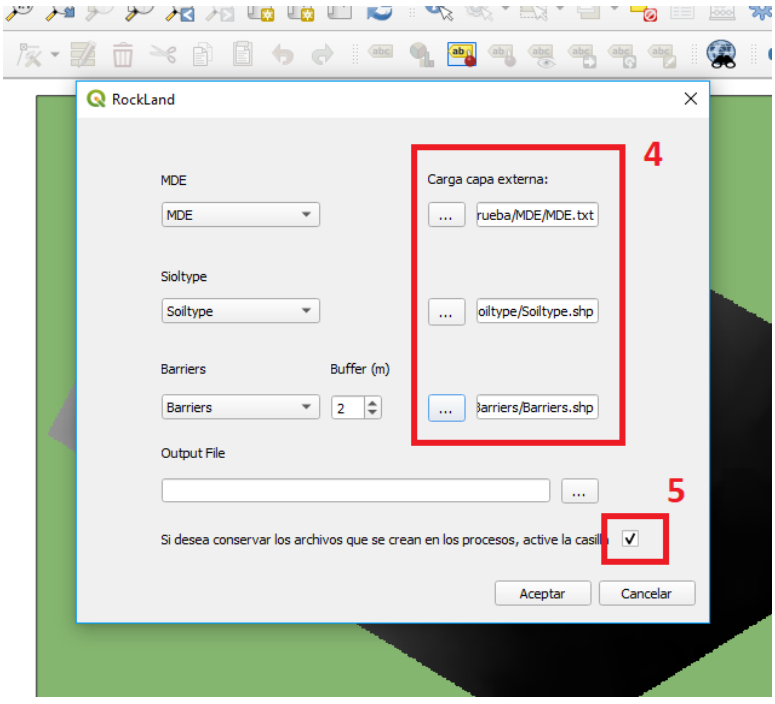

***Figura 25**. Funciones del plugin*

En el caso de la **Figura 25** el comportamiento del plugin es diferente, omite la capa seleccionada en el *comboBox* y prevalece la capa cargada en el *pushButton/lineEdit*, se presenta de esta manera para mayor versatilidad al plugin, si se olvida cargar una capa en la TOC, no hace falta insertarla, mediante esta opción se carga directamente en el plugin. También hay un comportamiento diferente cuando se activa el *checkBox* (Punto 5), los archivos intermedios creados se mantienen en el directorio seleccionado donde se guarda la capa de salida. (**Figura 26**)

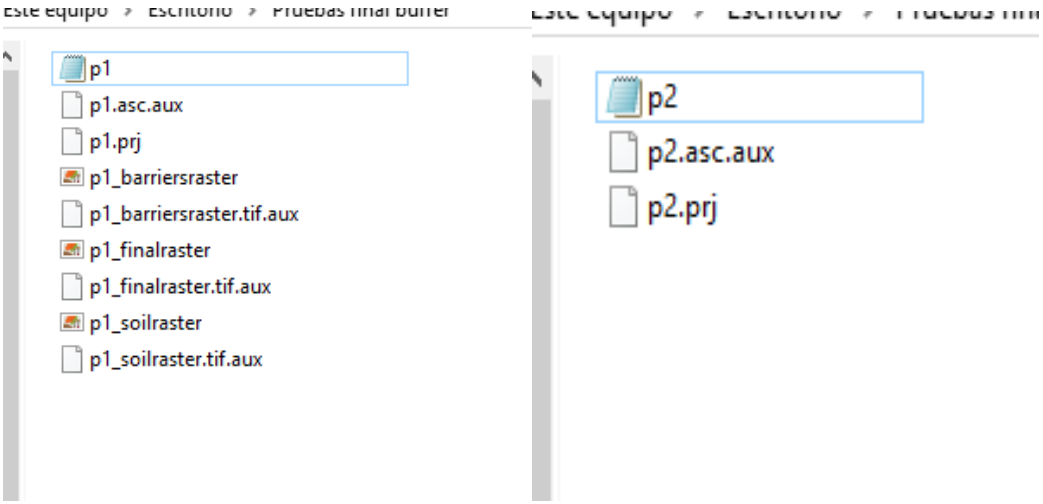

***Figura 26**. Diferencia entre activar el checkBox, a la derecha se muestran los archivos intermedios cuando el checkBox es activado, a la izquierda solo se muestra el archivo final de salida.*

También podemos hacer una combinación entre capas que se encuentran en la TOC y capas que se cargan externamente, como se muestra en la **Figura 27** (Punto 6), se comenta anteriormente que cuando cargamos una capa a través del *pushButton/lineEdit*, omite la capa que se encuentra en el *comboBox*, pues como se muestra en la imagen en el *comboBox* hay cargada una capa que no corresponde,pero si en el *pushButton/lineEdit* hay cargada la capa adecuada (Punto 6), el plugin se ejecuta correctamente.

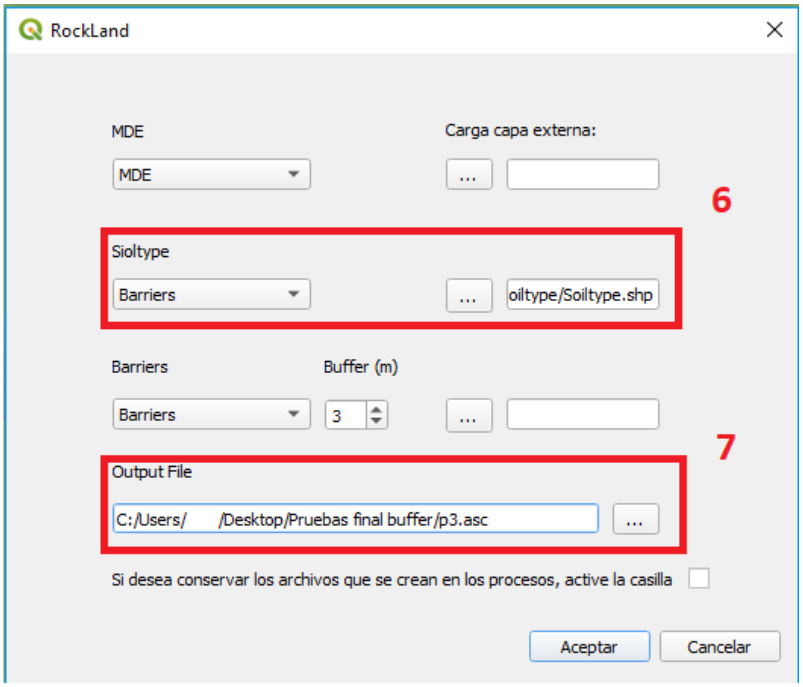

***Figura 27**. Funciones del plugin*

Finalmente se selecciona el directorio donde guardar la capa de salida (Punto 7), junto con los demás archivos si se desean mantener.

Por ultimo una vez aceptas la ejecución del plugin, aparece el widget de selección de sistema de referencia, al principio del apartado se menciona que no cal que todas las capas estén en el mismo sistema de referencia ya que seleccionándolo en el widget se aplica a todas las capas que interviene tanto de entrada como de salida. (**Figura 28**).

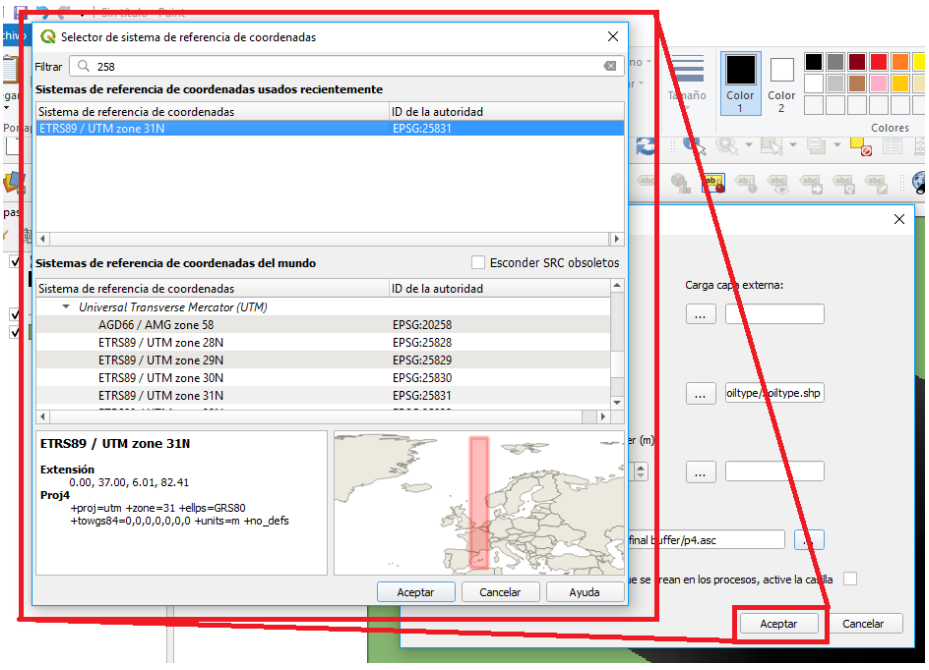

***Figura 28**. Muestra el Widget de selección del sistema de referencia*

En la **Figura 29** se muestra el archivo de salida que se genera finalmente al ejecutar el plugin, es el archivo necesario para realizar simulaciones, en estas se tiene en cuenta el tipo de suelo por el que recorren su trayectoria, ya que no tienen el mismo comportamiento, en un rodamiento con vegetación que, sin ella, esta puede afectar a la velocidad de deslizamiento y son cosas que se tienen en cuenta a la hora de realizar el estudio. Por otro lado, está la presencia de barreras protectoras, que tienen la función de evitar que las rocas lleguen a lugares que puedan crear un impacto mayor, como pueden ser carreteras o zonas residenciales. El archivo generado en este proyecto recoge estos aspectos.

***Figura 29**. Archivo Final, combinación entre los datos de entrada*

7 CONCLUSIONES

En el desarrollo del actual TFG se ha profundizado en la automatización de procesos a desarrollar en el software QGIS. Para lo cual se ha tenido que analizar la forma en que se accede y trabaja con las diferentes capas de información, para entender y manejar adecuadamente todas las posibilidades que ofrecen. Desde la función más básica, como representar los objetos espaciales, hasta la creación de scripts dentro de estos que nos permiten crear aplicaciones para ampliar las funciones según la necesidad.

Como punto final al presente trabajo se ha conseguido el objetivo que se requería, crear e implementar un complemento dentro del software QGIS, la cual permite automatizar un proceso. En este caso la creación de un archivo que contiene información geoespacial, sería muy tedioso tener que crear cada archivo manualmente. A partir de una información previa se consigue obtener otra distinta o en este caso una combinación, por consiguiente, se ahorra tiempo y por lo tanto dinero. En este proceso cabe resaltar la importancia de utilizar la tecnología de código libre. Es importante también compartir las herramientas que cada usuario genera de forma que también puedan ser utilizadas por otras personas, de este modo todo el mundo resulta beneficiado.

A nivel personal, el proyecto me ha ayudado a dar un paso adelante en los conocimientos previos que disponía sobre SIG, a entender mejor su estructura y a saber manejarla apropiadamente, para futuros proyectos. Creo poder gestionar futuros proyectos gracias a la habilidad adquirida en este.

Por otro lado, a pesar de la gran comunidad, encontrar información relevante, ordenada y clara, se hace difícil, en mi caso ha habido muchas pruebas de ensayo y error, y mucha dedicación a la investigación de procesos y algoritmos idóneos para el desarrollo del complemento, gracias a ello me ha llevado a entender bien cada paso que se ha realizado en el mismo. Cabe destacar también la importancia de documentar bien, los pasos seguidos, para que otras personas puedan acceder a la información.

8 BIBLIOGRAFIA

[1] Matas G., Lantada N., Corominas J. Gili J.A., Ruiz-Carulla R., Prades A. (2017). RockGIS: A GIS-based model for the analysis of fragmentation in rockfalls. Landslides 14(5), 1565-1578. DOI.:10.1007/s10346-017-0818-7. [Consulta 03/2019]
Disponible en:
http://dx.doi.org/10.1007/s10346-017-0818-7

[2] Olaya V., Sistemas de Información Geográfica [En línea]. España,2014. Licencia: Creative Common Atribución. [Consulta 17/03/2019].

Disponible en: http://volaya.github.io/libro-sig/

[3] QGIS (https://es.wikipedia.org/wiki/QGIS). [Consulta 04/2019]

[4] Python (https://es.wikipedia.org/wiki/Python). [Consulta 04/2019]

[5] Qt Designer Manual (https://doc.qt.io/qt-5/qtdesigner-manual.html) . [Consulta 04/2019]

[6] Datos espaciales (https://www.ager.es/productos/gis/datos.php). [Consulta 06/2019]

[7] GDAL (https://es.wikipedia.org/wiki/GDAL). [Consulta 06/2019]

[8] Corominas J, Lantada N., Gili JA, Ruiz-Carulla R., Matas G., Mavrouli A., Núñez-Andrés MA, Moya J., Buill F., Abellan A., Puig C., Prades A., Martinez-Bofill J., Saló L. (2017). The RockRisk Project: Rockfall risk quantification and prevention**.** 6th Interdisciplinary Workshop on Rockfall Protection, RocExs 2017 (Barcelona, España) [En línea] [Consulta 27/03/2019]

[9]Hinojosa Gutiérrez, A.P. PYTHON paso a paso.Madrid:Ra-Ma Editorial,2016.ISBN: 978-84-9964-611-4

[10]Lawhead, J. QGIS Python Programming Cookbook[En línea].Birmingham:Packt Publishing,2015.ISBN: 978-1-78398-498-5.
[Consulta: 20 Septiembre 2018].
Disponible a: http://www.green-forums.info/greenlib/geolibrary/Lawhead%20J/QGIS%20Python%20Programming%20Cookbook.%2020%20%2852%29/QGIS%20Python%20Programming%20Cookboo%20-%20Lawhead%20J.pdf

[11]Bahit, E. Curso:Python para Principiantes[En línea].Buenos Aires,Argentina:Licencia: Creative Commons Reconocimiento-NoComercial-SinObraDerivada 3.0,2012.Safe Creative:1207302042960. [Consulta 16 Marzo 2018].
Disponible a: (https://www.iaa.csic.es/python/curso-python-para-principiantes.pdf)

ANEJO I: CÓDIGO DEL COMPLEMENTO

```
# -*- coding: utf-8 -*-
"""
/***************************************************************************
 RockLand
                                 A QGIS plugin
 Creación de un raster con información combinada
 Generated by Plugin Builder: http://g-sherman.github.io/Qgis-Plugin-Builder/
                              -------------------
        begin                : 2019-03-31
        git sha              : $Format:%H$
        copyright            : (C) 2019 by Alejandro Quiroga
        email                : quiroga_.ap@hotmail.com

 ***************************************************************************/

/***************************************************************************
 *                                                                         *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or     *
 *   (at your option) any later version.                                   *
 *                                                                         *

 ***************************************************************************/
"""
from PyQt5 import*
from PyQt5.QtCore import QSettings, QTranslator, qVersion, QCoreApplication
from PyQt5.QtGui import QIcon,QColor
from PyQt5.QtWidgets import QAction, QFileDialog
import qgis
from qgis.core import *
from qgis.utils import *
from qgis.gui import *
from qgis.analysis import *
import processing
from os import *
import numpy

# Initialize Qt resources from file resources.py
from .resources import *
```

```
# Import the code for the dialog
from .RockTerreno_dialog import RockLandDialog
import os.path


class RockLand:
    """QGIS Plugin Implementation."""

    def __init__(self, iface):
        """Constructor.

        :param iface: An interface instance that will be passed to this class
            which provides the hook by which you can manipulate the QGIS
            application at run time.
        :type iface: QgsInterface
        """
        # Save reference to the QGIS interface
        self.iface = iface
        # initialize plugin directory
        self.plugin_dir = os.path.dirname(__file__)
        # initialize locale
        locale = QSettings().value('locale/userLocale')[0:2]
        locale_path = os.path.join(
            self.plugin_dir,
            'i18n',
            'RockLand_{}.qm'.format(locale))

        if os.path.exists(locale_path):
            self.translator = QTranslator()
            self.translator.load(locale_path)

            if qVersion() > '4.3.3':
                QCoreApplication.installTranslator(self.translator)

        # Declare instance attributes
        self.actions = []
        self.menu = self.tr(u'&RockLand')

        # Check if plugin was started the first time in current QGIS session
        # Must be set in initGui() to survive plugin reloads
        self.first_start = None


    # noinspection PyMethodMayBeStatic
    def tr(self, message):
        """Get the translation for a string using Qt translation API.
```

```
        We implement this ourselves since we do not inherit QObject.

        :param message: String for translation.
        :type message: str, QString

        :returns: Translated version of message.
        :rtype: QString
        """
        # noinspection PyTypeChecker,PyArgumentList,PyCallByClass
        return QCoreApplication.translate('RockLand', message)


    def add_action(
        self,
        icon_path,
        text,
        callback,
        enabled_flag=True,
        add_to_menu=True,
        add_to_toolbar=True,
        status_tip=None,
        whats_this=None,
        parent=None):
        """Add a toolbar icon to the toolbar.

        :param icon_path: Path to the icon for this action. Can be a resource
            path (e.g. ':/plugins/foo/bar.png') or a normal file system path.
        :type icon_path: str

        :param text: Text that should be shown in menu items for this action.
        :type text: str

        :param callback: Function to be called when the action is triggered.
        :type callback: function

        :param enabled_flag: A flag indicating if the action should be enabled
            by default. Defaults to True.
        :type enabled_flag: bool

        :param add_to_menu: Flag indicating whether the action should also
            be added to the menu. Defaults to True.
        :type add_to_menu: bool

        :param add_to_toolbar: Flag indicating whether the action should also
            be added to the toolbar. Defaults to True.
        :type add_to_toolbar: bool
```

```
        :param status_tip: Optional text to show in a popup when mouse pointer
            hovers over the action.
        :type status_tip: str

        :param parent: Parent widget for the new action. Defaults None.
        :type parent: QWidget

        :param whats_this: Optional text to show in the status bar when the
            mouse pointer hovers over the action.

        :returns: The action that was created. Note that the action is also
            added to self.actions list.
        :rtype: QAction
        """

        icon = QIcon(icon_path)
        action = QAction(icon, text, parent)
        action.triggered.connect(callback)
        action.setEnabled(enabled_flag)

        if status_tip is not None:
            action.setStatusTip(status_tip)

        if whats_this is not None:
            action.setWhatsThis(whats_this)

        if add_to_toolbar:
            # Adds plugin icon to Plugins toolbar
            self.iface.addToolBarIcon(action)

        if add_to_menu:
            self.iface.addPluginToMenu(
                self.menu,
                action)

        self.actions.append(action)

        return action

    def initGui(self):
        """Create the menu entries and toolbar icons inside the QGIS GUI."""

        icon_path = ':/plugins/RockTerreno/icon.png'
        self.add_action(
            icon_path,
            text=self.tr(u''),
            callback=self.run,
```

```
            parent=self.iface.mainWindow())

        # will be set False in run()
        self.first_start = True


    def unload(self):
        """Removes the plugin menu item and icon from QGIS GUI."""
        for action in self.actions:
            self.iface.removePluginMenu(
                self.tr(u'&RockLand'),
                action)
            self.iface.removeToolBarIcon(action)

    def sistemareferencia(self):
            CRSmap=QgsProjectionSelectionDialog()
            CRSmap.exec_()
            CRS=CRSmap.crs()
            return CRS

    def soiltypeR (self,mde,soiltype,CRS,filename):
            mdeExtent=mde.extent()
            xmin=mdeExtent.xMinimum()
            print(mdeExtent)
            xmax=mdeExtent.xMaximum()
            ymin=mdeExtent.yMinimum()
            ymax=mdeExtent.yMaximum()
            mdeWidth=mde.width()
            mdeHeight=mde.height()
            mdePixelX=mde.rasterUnitsPerPixelX()
            mdePixelY=mde.rasterUnitsPerPixelY()
            #Parametros que necesitamos para ejecutar el proceso
            pathsoilraster=filename[:-4] + '_soilraster.tif'
            params={"INPUT" :
soiltype,"FIELD":'Solitype',"UNITS":1,"WIDTH":mdePixelX,"HEIGHT":mdePixelY,"EXTEN
T" : "%f,%f,%f,%f"% (xmin, xmax, ymin,
ymax),"NODATA":0,"DATA_TYPE":6,"INVERT":0,"OUTPUT" : pathsoilraster}
            #Ejecutamos la rasterizacion
            rasterize = processing.run('gdal:rasterize', params)
            soilraster=QgsRasterLayer(pathsoilraster,'soilraster')
            #soilraster=QgsRasterLayer(r'C:\Users\quiro\Desktop\Pruebas final
buffer\soilraster.tif','soilraster')
            soilraster.setCrs(CRS)
            QgsProject.instance().addMapLayer(soilraster)
            return soilraster
```

```
    def barriersR(self,mde,barriers,tamañobuffer,CRS,filename):
        mdeExtent=mde.extent()
        #print (mdeExtent)
        #Guardamos los valores de la extension en variables necesarias para el algoritmo
        xmin=mdeExtent.xMinimum()
        xmax=mdeExtent.xMaximum()
        ymin=mdeExtent.yMinimum()
        ymax=mdeExtent.yMaximum()
        #print (xmin,xmax,ymin,ymax)
        #Ancho y alto de la capa MDE
        mdeWidth=mde.width()
        #print (mdeWidth)
        mdeHeight=mde.height()
        #print (mdeHeight)
        #Tamaño de pixel de la capa MDE
        mdePixelX=mde.rasterUnitsPerPixelX()
        #print (mdePixelX)
        mdePixelY=mde.rasterUnitsPerPixelY()
        #print (mdePixelY)
        pathbarriersraster=filename[:-4] + '_barriersraster.tif'
        if tamañobuffer == 0:
            #Parametros que necesitamos para ejecutar el proceso
            params={"INPUT" :
barriers,"FIELD":'id',"UNITS":1,"WIDTH":mdePixelX,"HEIGHT":mdePixelY,"EXTENT" :
"%f,%f,%f,%f"% (xmin, xmax, ymin,
ymax),"NODATA":0,"DATA_TYPE":6,"INVERT":0,"OUTPUT" : pathbarriersraster}
            #Ejecutamos la rasterizacion
            rasterize = processing.run('gdal:rasterize', params)
        else:
            attr = barriers.dataProvider().fields().toList()
            temp = QgsVectorLayer("Polygon?crs=epsg:25831", "result", "memory")
            temp.dataProvider().addAttributes(attr)
            temp.updateFields()
            features=QgsFeature()

            for feat in barriers.getFeatures():
                    features.setGeometry(feat.geometry().buffer(tamañobuffer,-1))
                    features.setAttributes(feat.attributes())
                    temp.dataProvider().addFeatures([features])
            #QgsProject.instance().addMapLayer(temp)
            #Parametros que necesitamos para ejecutar el proceso
            paramsb={"INPUT" :
temp,"FIELD":'id',"UNITS":1,"WIDTH":mdePixelX,"HEIGHT":mdePixelY,"EXTENT" :
"%f,%f,%f,%f"% (xmin, xmax, ymin,
ymax),"NODATA":0,"DATA_TYPE":6,"INVERT":0,"OUTPUT" : pathbarriersraster}
            #Ejecutamos la rasterizacion
            rasterize = processing.run('gdal:rasterize', paramsb)
```

```
        #Añadimos el nuevo raster al proyecto
        barriersRaster=QgsRasterLayer(pathbarriersraster,'barriersRaster')
        barriersRaster.setCrs(CRS)
        #!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
        barriersRaster.dataProvider().setNoDataValue(1,999)
        QgsProject.instance().addMapLayer(barriersRaster)
        return barriersRaster


    def F3(self,mde,soiltype,barriers,soilraster,barriersRaster,filename):
            entries=[]
            f1=QgsRasterCalculatorEntry()
            f1.raster=soilraster
            f1.ref='f1@1'
            f1.bandNumber=1
            entries.append(f1)

            f2=QgsRasterCalculatorEntry()
            f2.raster=barriersRaster
            f2.ref='f2@1'
            f2.bandNumber=1
            entries.append(f2)
            #print (entries)
            pathfinalraster=filename[:-4] + '_finalraster.tif'
            #((f2@1>0)*f2@1)+((f2@1=0)*f1@1)

calc=QgsRasterCalculator('((f2@1>0)*f2@1)+((f2@1=0)*f1@1)',pathfinalraster,'GTiff',
mde.extent(),mde.width(),mde.height(),entries)
            calc.processCalculation()

            final=QgsRasterLayer(pathfinalraster,'final')

            #final.setCrs(CRS)
            # Le colocamos una rampa de color para su visulizacion
            s=QgsRasterShader()
            c=QgsColorRampShader()
            soilcampos=soiltype.dataProvider().fields().toList()
            for fieldsoil in soilcampos:
              if fieldsoil.typeName() == 'Integer':

soilmax=soiltype.maximumValue(soiltype.fields().lookupField(fieldsoil.name()))

soilmin=soiltype.minimumValue(soiltype.fields().lookupField(fieldsoil.name()))
            barrcampos=barriers.dataProvider().fields().toList()
            for fieldbarr in barrcampos:
              if fieldbarr.typeName() == 'Integer64':
```

```
barrmax=barriers.maximumValue(barriers.fields().lookupField(fieldbarr.name()))

barrmin=barriers.minimumValue(barriers.fields().lookupField(fieldbarr.name()))
            i=[]
            i.append(QgsColorRampShader.ColorRampItem(soilmin,
QColor('#3fff19'),str(soilmin)))
            i.append(QgsColorRampShader.ColorRampItem(soilmax,
QColor('#217a0b'),str(soilmax)))
            i.append(QgsColorRampShader.ColorRampItem(barrmin,
QColor('yellow'),str(barrmin)))
            i.append(QgsColorRampShader.ColorRampItem(barrmax,
QColor('red'),str(barrmax)))
            c.setColorRampType(QgsColorRampShader.Interpolated)
            c.setColorRampItemList(i)
            s.setRasterShaderFunction(c)
            ps=QgsSingleBandPseudoColorRenderer(final.dataProvider(),1,s)
            final.setRenderer(ps)
            QgsProject.instance().addMapLayer(final)
            return final

    def Convert(self,final,filename,CRS):
        datatype=final.type()

        bufferParams={"INPUT":final,"TARGET_CRS":CRS,"NODATA":0,"COPY_SUBDATA
SETS":True,"DATA_TYPE":datatype,"OUTPUT":filename}
        processing.run('gdal:translate',bufferParams)

    def select_input_file(self):
        inputfile, _filter=QFileDialog.getOpenFileName(self.dlg,"Select input file","","All
Files (*);;Text Files (*.txt)")
        self.dlg.lineEdit.setText(str(inputfile))
    def select_input_file2(self):
        inputfile2, _filter=QFileDialog.getOpenFileName(self.dlg,"Select input
file","","All Files (*);;Shape Files (*.shp)")
        self.dlg.lineEdit_2.setText(inputfile2)
    def select_input_file3(self):
        inputfile3, _filter=QFileDialog.getOpenFileName(self.dlg,"Select input
file","","All Files (*);;Shape Files (*.shp)")
        self.dlg.lineEdit_3.setText(inputfile3)


    def select_output_file(self):
        filename, _filter = QFileDialog.getSaveFileName(self.dlg,"Select output file
","","*.asc")
        self.dlg.lineEdit_4.setText(filename)
```

```
    def remove(self,filename,soilraster,barriersRaster,final,borrafiles):
            QgsProject.instance().removeMapLayer(soilraster)
            QgsProject.instance().removeMapLayer(barriersRaster)
            QgsProject.instance().removeMapLayer(final)
            if borrafiles==False:
                    pathsoilraster=filename[:-4] + '_soilraster.tif'
                    os.remove(pathsoilraster)
                    os.remove(pathsoilraster + '.aux.xml')

                    pathbarriersraster=filename[:-4] + '_barriersraster.tif'
                    os.remove(pathbarriersraster)
                    os.remove(pathbarriersraster + '.aux.xml')

                    pathfinalraster=filename[:-4] + '_finalraster.tif'
                    os.remove(pathfinalraster)
                    os.remove(pathfinalraster + '.aux.xml')


    def run(self):
        """Run method that performs all the real work"""

        # Create the dialog with elements (after translation) and keep reference
        # Only create GUI ONCE in callback, so that it will only load when the plugin is
started
        if self.first_start == True:
            self.first_start = False
            self.dlg = RockLandDialog()
            # da funcionalidad al boton
            self.dlg.pushButton.clicked.connect(self.select_input_file)
            self.dlg.pushButton_2.clicked.connect(self.select_input_file2)
            self.dlg.pushButton_3.clicked.connect(self.select_input_file3)
            self.dlg.pushButton_4.clicked.connect(self.select_output_file)

        #vacia la caja de lineedit donde se especifica el nombre del archivo
        self.dlg.lineEdit.clear()
        self.dlg.lineEdit_2.clear()
        self.dlg.lineEdit_3.clear()
        self.dlg.lineEdit_4.clear()

        self.dlg.spinBox.setValue(0)
                    #conectamos el combox con las lineas

        # show the dialog
        self.dlg.show()
        #Vacia el comboBox ya que se acuulan las entradas
        while self.dlg.comboBox.count() > 0:
            self.dlg.comboBox.removeItem(0)
```

```
        while self.dlg.comboBox_2.count() > 0:
            self.dlg.comboBox_2.removeItem(0)
        while self.dlg.comboBox_3.count() > 0:
            self.dlg.comboBox_3.removeItem(0)

        #Lee las capas que se encuentran en la TOC (Tabla de contenidos)
        #y añadimos las capas a los comboBox
        #layers = qgis.core.QgsProject.instance().layerTreeRoot().children()
        '''self.dlg.comboBox.addItems([layer.name() for layer in layers if
layer.layer().type()==QgsMapLayer.RasterLayer])
        self.dlg.comboBox_2.addItems([layer.name() for layer in layers if
layer.layer().type()!=QgsMapLayer.RasterLayer])
        self.dlg.comboBox_3.addItems([layer.name() for layer in layers if
layer.layer().type()!=QgsMapLayer.RasterLayer])'''
        layers=qgis.core.QgsProject.instance().layerTreeRoot().children()
        listR=[layer.name() for layer in layers if
layer.layer().type()==QgsMapLayer.RasterLayer]

        listaR=[]
        indicesR=[]
        listaV=[]
        indicesV=[]
        i=0
        for layer in layers:
            if layer.layer().type()==QgsMapLayer.RasterLayer:
                listaR.append(layer.name())
                indicesR.append(i)
            else:
                listaV.append(layer.name())
                indicesV.append(i)
            i+=1

        self.dlg.comboBox.addItems(listaR)
        #QgsProject.instance().layerTreeRoot.insertLayer(0,listR)
        listV=[layer.name() for layer in layers if
layer.layer().type()!=QgsMapLayer.RasterLayer]
        self.dlg.comboBox_2.addItems(listaV)
        self.dlg.comboBox_3.addItems(listaV)



        # Run the dialog event loop
        result = self.dlg.exec_()
        # See if OK was pressed
        if result:
            # Do something useful here - delete the line containing pass and
            # substitute with your code.
```

```
        filename=self.dlg.lineEdit_4.text()
        inputfile=self.dlg.lineEdit.text()
        inputfile2=self.dlg.lineEdit_2.text()
        inputfile3=self.dlg.lineEdit_3.text()

        """
        #selectedLayerIndex=self.dlg.comboBox.currentIndex()
        selectedLayerIndex=self.dlg.comboBox.currentIndex()
        capasenlalista=self.dlg.comboBox.count()
        selectedLayerIndex2=self.dlg.comboBox_2.currentIndex()+capasenlalista
        selectedLayerIndex3=self.dlg.comboBox_3.currentIndex()+capasenlalista
        aaa=layers[selectedLayerIndex].name()
        print(aaa)

        print(layers[selectedLayerIndex].layer().name(),selectedLayerIndex)
        print(layers[selectedLayerIndex2].layer().name(),selectedLayerIndex2)
        print(layers[selectedLayerIndex3].layer().name(),selectedLayerIndex3)
        """


        mdepath=layers[indicesR[self.dlg.comboBox.currentIndex()]].layer().source()
        print(mdepath)

soiltypepath=layers[indicesV[self.dlg.comboBox_2.currentIndex()]].layer().source()
        print(soiltypepath)

barrierspath=layers[indicesV[self.dlg.comboBox_3.currentIndex()]].layer().source()
        print(barrierspath)

        """
        mdepath=layers[selectedLayerIndex].layer().source()
        print(mdepath)
        soiltypepath=layers[selectedLayerIndex2].layer().source()
        print(soiltypepath)
        barrierspath=layers[selectedLayerIndex3].layer().source()
        print(barrierspath)
        """

        if inputfile == "":
         mde=QgsRasterLayer(mdepath,'mde')
        else:
         mde=QgsRasterLayer(inputfile,'mde')
        print(mde)
        if inputfile2 == "":
         soiltype=QgsVectorLayer(soiltypepath,'soiltype','ogr')
        else:
         soiltype=QgsVectorLayer(inputfile2,'soiltype','ogr')
```

```
        if inputfile3 == "":
         barriers=QgsVectorLayer(barrierspath,'barriers','ogr')
        else:
                barriers=QgsVectorLayer(inputfile3,'barriers','ogr')

        tamañobuffer=self.dlg.spinBox.value()
        CRS=self.sistemareferencia()
        QgsProject.instance().setCrs(CRS)
        soilraster=self.soiltypeR(mde,soiltype,CRS,filename)
        barriersRaster=self.barriersR(mde,barriers,tamañobuffer,CRS,filename)
        final=self.F3(mde,soiltype,barriers,soilraster,barriersRaster,filename)
        self.Convert(final,filename,CRS)
        borrafiles=self.dlg.checkBox.isChecked()
        self.remove(filename,soilraster,barriersRaster,final,borrafiles)
        self.iface.messageBar().pushMessage("Success", "Output file written at " +
filename, level=Qgis.Success, duration=5)
```